The Waite Group

PLUME
WAITE

HIDDEN POWERS OF THE TRS-80® MODEL 100

ACCESS TO ADVANCED PROGRAMMING FEATURES

- Call up powerful hidden routines from BASIC or assembler
- Includes in-depth explanations of ROM operation, peripherals, and BASIC
- Enables you to create professional-quality programs
- Explains undocumented features

by
Christopher L. Morgan

Another book by the bestselling authors of *ASSEMBLY LANGUAGE PRIMER FOR THE IBM® PC & XT* and *BLUE-BOOK OF ASSEMBLY ROUTINES FOR THE IBM® PC & XT*. . . . And rave reviews for The Waite Group:

"An outstanding example of how to write a technical book for the beginner . . . refreshingly enjoyable . . . accurate, readable, understandable, and indispensable. Don't stay home without it."

— Ken Barber, reviewing *CP/M Primer*, in *Microcomputing*

"Mitch Waite . . . has left a distinctive contribution to the literature of computer graphics . . . seeing it here is like understanding it for the first time."

— *Computer Graphics Primer*, reviewed in *Computer Graphics World*

"It's hard to imagine that a field only a decade old already has a classic, but Waite's book is just that."

— Tony Dirksen, reviewing *Computer Graphics Primer* in *Interface Age*

". . . an outstanding reference work, aside from its obvious benefit as an instructional text . . . superb writing style . . ."

— Chuck Dougherty, reviewing *Soul of CP/M* in *Cider*

"The demystification of a complex technological development . . . rating: 100."

— *8086/8088 16-bit Microprocessor Primer*, reviewed in *The Reader's Guide to Microcomputer Books*

"[This book] can have anybody writing and understanding assembly language programs within one hour."

— Alan Neibauer, reviewing *Soul of CP/M*, in *80 Microcomputing*

"All-inclusive, beautifully organized, easy-to-use reference that helps you wrest order from chaos . . ."

— *CP/M Bible*, reviewed in *Byte Book Club Bulletin*

Christopher L. Morgan

Christopher L. Morgan is a professor at California State University, Hayward, where he teaches mathematics and computer science, including computer graphics, assembly language programming, computer architecture, and operating systems. Dr. Morgan has given talks and authored papers in pure mathematics and on representations of higher-dimensional objects on computers. He is director of the computer graphics lab at Hayward and is a member of a number of professional associations, including the American Mathematical Society, the National Council of Teachers of Mathematics, and the Association for Computing Machinery. He is coauthor along with Mitchell Waite of *8086/8088 16-Bit Microprocessor Primer* and *Graphics Primer for the IBM® PC*. He is the author of *Bluebook of Assembly Language Routines for the IBM® PC and XT*.

HIDDEN POWERS OF THE TRS-80® MODEL 100

by Christopher L. Morgan

A Plume/Waite Book
New American Library
New York and Scarborough, Ontario

PLUME TRADEMARK REG. U.S. PAT. OFF. AND FOREIGN COUNTRIES
REGISTERED TRADEMARK — MARCA REGISTRADA
HECHO EN WESTFORD, MASS., U.S.A.

SIGNET, SIGNET CLASSIC, MENTOR, PLUME, MERIDIAN and NAL BOOKS are published *in the United States* by New American Library, 1633 Broadway, New York, New York 10019, *in Canada* by The New American Library of Canada Limited, 81 Mack Avenue, Scarborough, Ontario M1L1M8

Cover design by Michael Manwaring
Illustrations by Winston and Karen Sin
Typography by Walker Graphics

First Printing, November, 1984

1 2 3 4 5 6 7 8 9

PRINTED IN THE UNITED STATES OF AMERICA

Contents

Acknowledgments vii
Preface viii

1 *Exploring the Model 100* 1

Overview of the Model 100 1
Peeking Inside the Model 100 4
Notes on Using This Book 16
Summary 18

2 *The Model 100's Hardware* 19

Why Study Hardware? 19
Hardware Overview 20
The 80C85 Central Processor 22
The System's Buses 24
ROM and RAM 26
Port Decoding 28
8155 Parallel Input/Output Interface Controller 31
Special Control Port 33
μPD 1990AC Real Time Clock 34
6402 Universal Asynchronous Receiver Transmitter 34
Liquid Crystal Display Screen 34
Keyboard 35
Printer Interface 35
Summary 37

3 *Hidden Powers of the ROM* 38

The Interrupt Entry Points 41
The BASIC Interpreter 51
The TELCOM Program 69
The MENU Program 71
The ADDRSS and SCHEDL Programs 77
The TEXT Program 78
The Initialization Routines 80
The Primitive Device Routines 81
Summary 81

4 *Hidden Powers of the Liquid Crystal Display* 82

How Liquid Crystal Displays Work 82
How to Program the LCD 88
ROM Routines for the LCD 91
Summary 114

5 *Hidden Powers of the Real-Time Clock* 115

How the Real-Time Clock Works 116
The ROM Routines 121
The Clock-Cursor-Keyboard Background Task 142
Summary 147

6 Hidden Powers of the Keyboard 148

How a Scanning Keyboard Works 148
How to Program the Model 100 Keyboard 149
Descriptions of ROM Routines 153
The Clock-Cursor-Keyboard Background Task 156
The Keyboard Input Routines 165
Summary 168

7 Hidden Powers of the Communications Devices 169

How an RS-232 Serial Communications Line Works 170
The RS-232C Connector and the Modem 174
The ROM Routines for the Communications Devices 174
Dialing the Telephone 180
Reading from the Serial Communications Line 185
Writing to the Serial Communications Line 191
Summary 195

8 Hidden Powers of Sound 196

How Sound Works in the Model 100 196
The ROM Routines For Sound 198
Summary 202

9 Hidden Powers of the Cassette 203

How the Cassette Interface Works 203
The ROM Routines for the Cassette System 205
Summary 217

Appendices 218

A. BASIC Function Addresses 218
B. BASIC Keywords 219
C. BASIC Command Addresses 223
D. Operator Priorities for Binary Operations 225
E. Some Numerical Conversion Routines 226
F. Binary Operations for Double Precision 226
G. Binary Operations for Single Precision 227
H. Binary Operations for Integers 227
I. Error Codes 228
J. BASIC Error Routines 229
K. Control Characters for the Model 100 230
L. Routines for Escape Sequence 231
M. Special Screen Routines for the Model 100 233
N. LCD Data for Character Positions 234
O. ASCII Table for Regular Keys 237
P. ASCII Table for NUM Key 241
Q. ASCII Table for Special Keys 242

Index 243

Acknowledgments

There are a number of people I would like to thank for their valuable help in making this book possible.

At the Waite Group, Robert Lafore has provided support, encouragement, guidance, and feedback which made my job at least an order of magnitude easier; Lyn Cordell has done a superb job in managing the production of the book; and Mitchell Waite initiated and oversaw the entire operation.

I would like to thank my wife Carol and my children Elizabeth and Thomas for their patience with me while I worked on this book.

Preface

This book is for anyone who wants to understand the inner secrets of the TRS-80® Model 100 portable computer. It will be equally useful to those who need to learn about the Model 100 in particular and to those who want to increase their general understanding of computers. This book and the Model 100 will provide a new world of fascinating study, and they'll both fit in your briefcase!

The Model 100 is the first of a new breed of lap-size computers that have launched a revolution in the way computing is done. No longer are we chained to our desks by a heavy machine requiring a wall plug and cumbersome peripherals. Now we can slip a full-fledged computer into a briefcase and take it with us on the airplane, to a client's office, or to a construction site.

Hidden Powers of the TRS-80® Model 100 reveals how this amazing machine works, on a level seldom glimpsed by the casual user or programmer. Using simple, down-to-earth language, this book explains the computer's hardware and the built-in software that make the Model 100 so powerful for its diminutive size.

Who Can Profit from This Book

If you are a programmer looking for ways to enhance your programs — to give them professional polish and speed — this book will show you how to use the powerful *undocumented routines* built into the Model 100's Read Only Memory. These routines can be accessed from either assembly language or BASIC; this book shows you how. Accessing these routines directly will make your programs more powerful and versatile. For instance, you will learn how to perform selective scrolling (scrolling only a few lines of the display), how to shift the display to "reverse video" (white on black) for special effects, and how to dial the telephone directly from BASIC.

If you are a programmer, learning how such hardware devices as the Liquid Crystal Display and keyboard work will enable you to perform tasks that are ordinarily impossible, such as setting up real-time displays of complex data or detecting any number of keys pressed simultaneously on the keyboard. Uses for such techniques include the simulation of instrument panels and other machines, both for games and for more serious programs.

If you are a hardware designer who would like to market a peripheral device for the Model 100, you will find it essential to know how the Model 100's hardware works and how it interfaces with the outside world and with the built-in software. This book will tell you what you need to know in order to design equipment that will work successfully in the Model 100 environment.

Finally, you may simply be curious about how computers work. *Hidden Powers* is written in a simple, jargon-free style. Anyone who has some familiarity with programming should be able to understand it, and it can open up a whole new area of exploration for someone who is learning about computers. You'll find that the computer is not a simple dumb beast waiting for your typed-in program. Instead, it is a surprisingly intelligent and complex machine that will reward your study and investigation.

Because many of the principles and chips used in the Model 100 are common to other computers as well, this in-depth investigation will help you to acquire an understanding of computers in general. The techniques and information presented in this book are similar, except in detail, to those you would use to investigate most other modern computers.

What You Need to Know to Use This Book

Any programmer who is familiar with a higher-level language such as BASIC should be able to profit from this book. The general explanations of the hardware and of the built-in ROM routines require no knowledge of assembly language. Many of the routines described can be called directly from BASIC programs. Also, example programs are given in BASIC so that you can experiment with the operation of the hardware.

Use of the disassembler program for further investigation of the Model 100's built-in ROM routines will require some knowledge of 8085 assembly language and of the hexadecimal numbering system. If you have access to a CP/M® system, a good book for learning about 8085 assembly language is *Soul of CP/M*, by Mitchell Waite and Robert Lafore (Indianapolis: Howard Sams & Co. Inc., 1983).

What's in This Book

The first few chapters in *Hidden Powers* provide some tools and background. You'll need these to understand the more detailed explanations of the individual components of the Model 100 that follow in later chapters.

Chapter 1 gives an overview of the Model 100, a quick once-over of the various elements that make up its hardware and software. Then some programming tools are introduced. The most important of these is a disassem-

bler. This program, written in BASIC, permits you to dig into the Model 100's ROM, to explore its every nuance and detail. In subsequent chapters, *Hidden Powers* provides entry points to and explains the use of all the most common ROM routines. With these as starting points, you can use the disassembler to investigate individual routines and determine exactly how they work. Other BASIC programs are provided to display key areas in the ROM routines and to search for specific patterns in the Model 100's memory.

Chapter 2 provides a detailed look at the architecture of the Model 100. The 80C85 Central Processing Unit is examined first, followed by the bus structure and the chips that handle input/output operations. If you are unfamiliar with computer hardware, you will find that this chapter opens the door to a new and fascinating world.

Chapters 3 through 9 investigate individual components of the Model 100: memory, display, real-time clock, keyboard, communications devices, sound, and cassette system. For each of these subjects the hardware is explained first. Then the ROM routines that control the hardware are explored in detail. Entry addresses and parameters are summarized for these routines so that BASIC and assembly language programs can call them directly.

Throughout these chapters short BASIC programs allow you to gain easy "hands on" experience with particular parts of the machine. These example programs should make many aspects of the Model 100's operation accessible even to those who are not assembly language programmers.

When you finish *Hidden Powers*, you'll have a thorough, detailed understanding of the Model 100 and its operation. If you have never investigated a computer in such depth before, you'll find that a fascinating new world has been opened to you. No longer will you be at the mercy of the incomprehensible "mysteries" in your computer: you'll have the technique and the knowledge to dig in, explore, and understand what's *really* going on in the Model 100 or almost any other computer system.

1

Exploring the Model 100

Concepts

Overview of the Model 100
Some tools for probing the Model 100
Notes on using this book

This chapter introduces the TRS-80 Model 100 Portable Computer, points you to the appropriate written material, and provides the essential software tools that will start you toward understanding the operation of the Model 100. There is also a discussion of the major features of this book, including its boxes describing the ROM routines and its programs written in BASIC.

The techniques and much of the information presented in this chapter and in the rest of the book will carry over to other computers, not only from Radio Shack but from other companies as well. This is because most microcomputers are organized on the same basic principles and use many of the same chips.

Overview of the Model 100

Let's take a quick look at the major features of the Model 100 to get our bearings before we make our first explorations into its inner structure.

The TRS-80 Model 100 Portable Computer is a truly remarkable piece of engineering. It is the size of a notebook, yet it is a full-fledged computer with facilities to perform the three major functions of modern information-handling systems: computing, text entry, and communications.

The 64K bytes of memory space in the Model 100 are divided into two halves. The lower 32K bytes consist of ROM (Read Only Memory), and the

upper 32K bytes contain RAM (Random Access Memory). You may have less RAM than this in your particular Model 100.

The Model 100's ROM houses its built-in programs. In Chapter 3 we will explore the entire contents of the ROM. Remarkably, the entire 32K of ROM is located on one chip.

The RAM contains user files (programs and text files) and the data needed to run the operating system. Not all 32K bytes of RAM need to be installed. The basic model comes with 8K bytes, and you can add more RAM in 8K increments until you fill the entire 32K bytes of available space.

The Model 100 uses a master menu to tie all its functions together. The master menu displays the names of the various programs and data files stored in the system. The file system resides entirely in the RAM and ROM memory of the computer, providing quick and easy access to a wide range of functions, features, and modes.

Unlike the situation with many computers, when you turn off the Model 100 in the normal manner (using the switch on the right side of the machine), the information in RAM is not lost. Thus both RAM and ROM behave like permanent memory, making secondary storage techniques such as cassette tape and disk not as vital as in many computers. You can store your favorite BASIC and machine-language programs in the main memory as long as you need them.

The file directory is stored in RAM, and the files it lists can be in either ROM or RAM. The Model 100 has five ROM files and as many as nineteen RAM files. Since the information in RAM does not disappear when the Model 100 is turned off, it makes sense to have this many RAM files active in the system at once. We will study the master menu and its directory in Chapter 3.

The programming power of the Model 100 comes from its Microsoft® BASIC interpreter. BASIC is the first item on the master menu. The BASIC in the Model 100 has the same syntax as the BASICs that come with most personal computers. However, because of size limitations, it is missing a few features. For example, there are no user-defined functions written in BASIC. Also, because of the memory-based file structure, saving and loading files is somewhat different — in some ways, better — than on disk-based systems, in that files are automatically updated as you work on them.

The code for the BASIC interpreter is contained in the first part of the Model 100's ROM. BASIC keeps its working data (variables, pointers, and buffers) in the highest part of memory, which is RAM. BASIC programs are stored throughout the rest of the Model 100's RAM along with other files (see Figure 1-1). In Chapter 3 we will explore the inner workings of the BASIC interpreter. You will see how BASIC interprets command lines, and you will learn where routines to perform all commands are located.

The Model 100 has its own built-in text editor, called TEXT. This is the second entry in the master menu. The code for TEXT is contained in ROM, above the code for BASIC. TEXT stores its variables and other kinds of working data in high memory in the same areas that BASIC uses for this purpose. In Chapter 3 we will also discuss how the TEXT program works.

A program called TELCOM allows you to move files conveniently between the Model 100 and other computers. The code for TELCOM resides in ROM, above the code for BASIC and below the code for TEXT. TELCOM uses the RS-232C serial port and the modem to send and receive files.

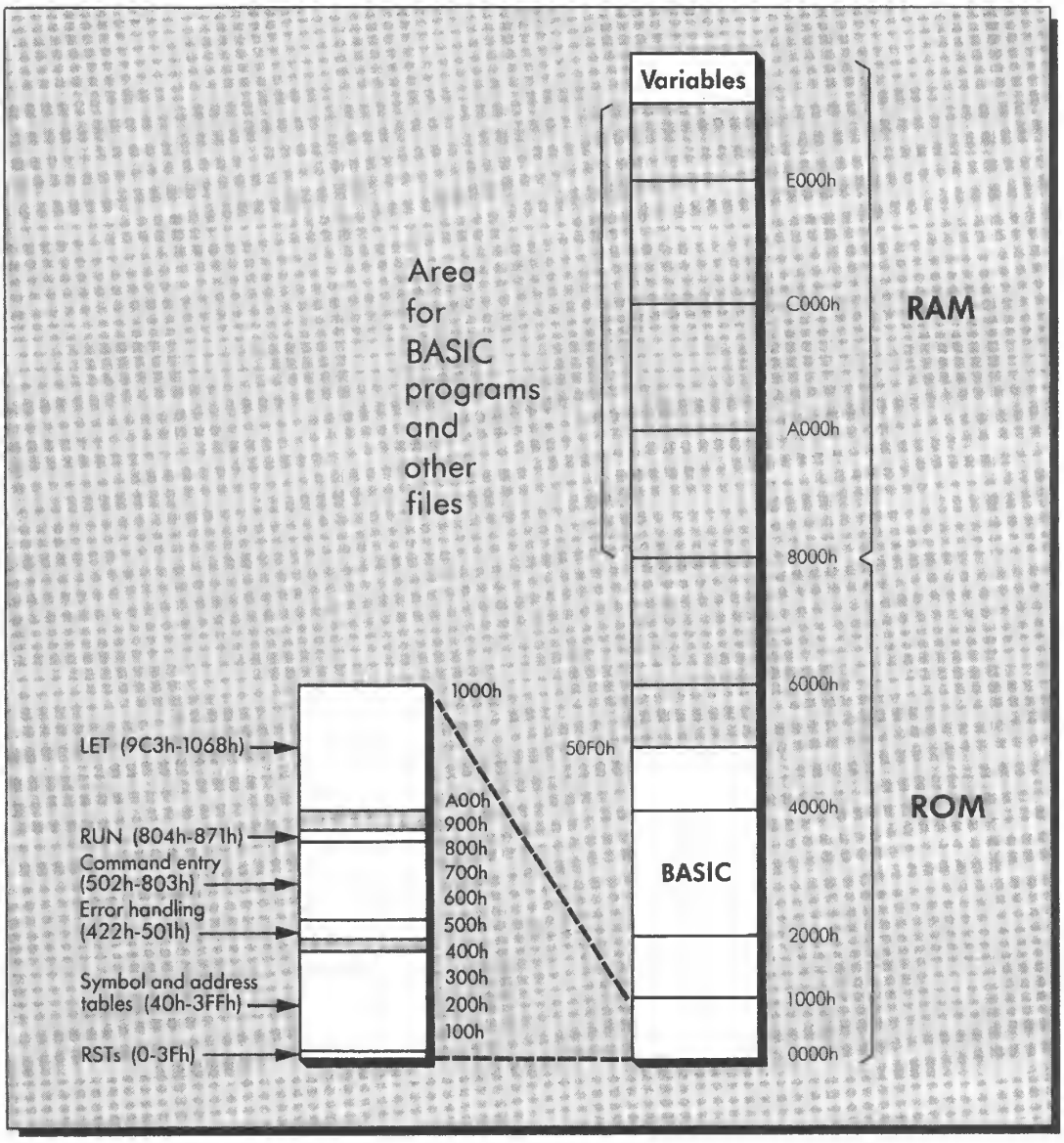


Figure 1-1. Memory map of BASIC

In Chapter 7 we will discuss these devices in detail, seeing how to program them directly, and in Chapter 3 we will examine the TELCOM program itself.

The Model 100 contains two other ROM programs, SCHEDL and ADDRSS. We will see how they fit into the scheme of things in Chapter 3.

The Model 100 has a wide variety of input and output devices, including a Liquid Crystal Display, a keyboard, a modem, an RS-232C serial port, a sound generator, a tape cassette interface, a printer interface, and a bar code reader interface. In Chapter 2 we will see how these various devices are arranged within the Model 100, and in Chapters 4 through 9 we will study many of these devices in detail.

Peeking Inside the Model 100

Now let's see how to find out what's going on inside the Model 100. In this section you'll learn how to get the Model 100 itself to tell you how it works by running BASIC programs on it.

Besides the Model 100 itself, two key sources of information about the machine's inner workings are the *TRS-80® Model 100* owner's manual and the *Radio Shack® Service Manual* for the Model 100. The owner's manual comes with the computer, and the *Service Manual* is available through Radio Shack stores and computer centers. A third source of information is the data sheets published by chip companies such as Intel, whose specifications and designs are used for some of the chips in the Model 100. These data sheets are compiled in books such as the *Component Data Catalog* from Intel®, available through their Literature Department as well as from some computer stores and electronic parts stores.

Memory Display

Let's begin exploring the Model 100 by writing a simple BASIC program that will display the contents of its memory. This display is a simple memory dump that shows the contents of the Model 100 as decimal values and as ASCII characters. Each line of output from this program displays six bytes of memory. This is the maximum number of bytes that can be displayed in this format. In each line the address of the first byte comes first, followed by the decimal value for each of the six bytes and finally by their corresponding ASCII characters. If a byte contains a control code such as carriage return, linefeed, or bell, a space is substituted for the character so that the display will not be affected.

```
100 ' DECIMAL/ASCII DUMP
110 '
120   FOR I = 0 TO 32767 STEP 6
130 '
140 ' ADDRESS
150    PRINT USING "#####";I;
160    PRINT "  ";
170 '
180 ' DECIMAL BYTES
190    FOR K = 0 TO 5
200     PRINT USING "####";PEEK(I+K);
210    NEXT K
220    PRINT " ";
230 '
240 ' ASCII CHARACTERS
250    FOR K = 0 TO 5
260     X = PEEK(I+K) AND 127
270     IF X<32 THEN X=32
280     PRINT CHR$(X);
290    NEXT K
300 '
310    PRINT
320   NEXT I
```

Looking at the program in detail, you can see that it consists of a FOR loop with index I that ranges from 0 to 32767, the entire length of the ROM. The STEP size for this loop is 6, corresponding to the fact that six bytes are displayed per line.

On line 140 the address of the first byte of the line is displayed. The PRINT USING statement formats the address to ensure that it always takes up the same number of spaces on the screen.

On lines 190-210, a FOR loop displays the decimal values of six bytes of memory. The PEEK function is used to fetch the contents of these bytes, and the PRINT USING command is used to ensure that their values appear evenly spaced on the screen.

On lines 250-290, a FOR loop displays the ASCII symbols corresponding to the same six bytes. On line 260, the PEEK function gets the value from memory. The value is ANDed with 127 to eliminate (make zero) the highest bit from the byte. Sometimes this bit is set (set to one) in the computer's memory to indicate when a character is the last letter of a name. We want to display characters as though the highest bit were not set; otherwise, we will get erroneous graphics characters when the highest bit is on. On line 270, the program checks for control codes and replaces them by the value 32, which is the ASCII code for a space. If we left the control codes

alone, the display would get messed up every once in a while. On line 280, the character is printed out using the CHR$ function.

Run this program now to see what's in the Model 100's memory. Starting at location 128, you should see the keywords corresponding to BASIC's commands. This is the first step in understanding how and where BASIC works. Notice that each keyword begins with a byte that has a high value (see the decimal values for these bytes). In fact, the numerical code for the initial letter of each keyword is equal to the ASCII code of the letter plus the value 128. This is the result of turning on (setting to 1) the highest bit of the byte. You can guess that BASIC uses this fact when it searches through this list of keywords.

After you run this program as it is, you can change the limits on line 120 and run it to explore other areas of memory. For example, if you look at high memory, you might discover where the directory is stored. Try replacing line 120 by:

```
120  FOR I = 63842 TO 64139 STEP 6
```

You should recognize the filenames in your directory as they go by. In Chapter 3, we'll show you a program that does a much better job at displaying the directory.

BASIC Keywords

Now let's look at BASIC's keywords in detail. The keywords form a complete guide to BASIC's commands and functions. Almost every BASIC command begins with a keyword that identifies the action to be done. Any command that doesn't start with a keyword is assumed to be a LET command; thus, even the commands that do not begin with a keyword have one assumed.

Here's a program that displays the keywords in tabular form. It lists all 127 BASIC keywords, numbered from 0 to 126.

```
100 ' COMMAND TABLE
110 '
120  FOR I = 128 TO 607
130   X=PEEK(I)
140   IF X<128 THEN 190
150   PRINT
160   PRINT USING "###";K;
170   K=K+1
180   PRINT " ";
190   PRINT CHR$(X AND 127);
200  NEXT I
```

The program loops through all the characters from location 128 to 607 (see line 120). You can find these limits by running the previous memory dump program. On line 130, the ASCII code of the character is fetched from memory. On line 140, the program in effect looks for values greater than or equal to 128, indicating the beginning of a keyword. If it finds such values, lines 150-180 are executed. They number the output lines. The variable K keeps track of the numbering. On line 190, the individual characters of the keywords are printed out.

The table produced by this program does several things. First of all, it gives a complete list of the keywords, even the ones that are not documented in the manual because they are not supported by Radio Shack. Second, it provides an ordering or numbering for the keywords. In Chapter 3 we'll see how this numbering is used to index into tables that give the address of the routines to execute BASIC commands and functions.

The Disassembler

The next step in our investigation of the Model 100 is to write a *disassembler*. A disassembler is a program that converts the raw machine language in the computer into assembly language that we can read. In this section we give a listing for a disassembler that is written in BASIC. This disassembler provides a window into the ROM, allowing us to see exactly how the Model 100 handles each command and controls each of its devices. Each routine discussed in this book will be identified by its address and a name. When you enter the address, the disassembler program will display the assembly code that starts there. You can use the PAUSE key to stop and start the display from the disassembler as you read through the discussion in this book. You can use the SHIFT BREAK keys to terminate the program so that you can start it up at a new address as the discussion shifts to a new location in the ROM.

When you run this program, it will ask you for a starting address and then an ending address. Both of these addresses must be entered as four-digit hexadecimal numbers. Each address in this book will be given in hexadecimal as well as in decimal representation so that you can always enter it directly into the disassembler program.

Once the starting and ending addresses have been entered, the program starts displaying the assembly-language equivalents of the machine language stored in the memory between these addresses. For each machine-language instruction the program will display a single line on the screen. On this line you will see the address of the first byte of the instruction, then the name (mnemonic) of the instruction, and finally any operands (data or address of the data).

A few words of warning about using any disassembler are in order. Not all of a computer's memory is filled with machine language. Even the ROM contains tables of data. If you try to disassemble memory locations that do not contain machine language, you will get nonsensical assembly language as your output. Generally, this is pretty easy to recognize when it occurs, and you can run the disassembler to find out where the various tables are located in ROM. However, it is often difficult to determine the exact place where machine language ends and data begins or vice versa. In particular, when your first address falls in the middle of a CPU instruction, the first few lines of output will often be incorrect.

You should type in the disassembler program and save it in your Model 100. You can save it as a RAM file, but you should also back it up by saving it on cassette tape or on the optional disk drive, or by uploading to another computer where it can be saved on disk.

Here is the disassembler.

```
100 ' PROGRAM DISASSEMBLER
110 '
120 ' THIS PROGRAM DISASSEMBLES
130 ' MACHINE CODE.  WHEN THE
140 ' PROGRAM SIGNS ON,
150 ' SPECIFY A STARTING ADDRESS
160 ' AND AN ENDING ADDRESS.
170 '
180   CLEAR 2000
190   GOTO 470 ' GOTO MAIN PROGRAM
200 '
210 ' SUBROUTINE -- HEXADECIMAL BYTE
220 R2=INT(A/16)  ' UPPER DIGIT
230 R1=A-R2*16    ' LOWER DIGIT
240 '
250 R2=R2+48:IF R2>57 THEN R2=R2+7
260 A$=CHR$(R2)
270 R1=R1+48:IF R1>57 THEN R1=R1+7
280 A$=A$+CHR$(R1)
290 RETURN
300 '
310 ' SUBROUTINE -- HEXADECIMAL WORD
320 J=I
330 Q2=INT(J/256):Q1=J-Q2*256
340 A=Q2:GOSUB 210:I$=A$
350 A=Q1:GOSUB 210:I$=I$+A$
360 RETURN
370 '
```

```
380 ' SUBROUTINE -- HEX WORD TO DEC
390 J=0
400 FOR K=1 TO 4
410 G=ASC(MID$(I$,K,1))-48
420 IF G>9 THEN G=G-7
430 J=16*J+G
440 NEXT K
450 RETURN
460 '
470 ' MAIN PROGRAM
480 '
490 DIM C$(256),L(256)
500 FOR I=0 TO 255:READ C$(I):NEXT I
510 FOR I=0 TO 255:READ L(I):NEXT I
520 '
530 INPUT "STARTING ADDRESS:";I$
540 GOSUB 380:I1=J
550 INPUT "ENDING ADDRESS  :";I$
560 GOSUB 380:I2=J:I=I1
570 '
580 ' TOP OF MAIN LOOP
590 GOSUB 310 ' HEX WORD
600 P$="H"+I$+" "
610 '
620 ' GET FIRST BYTE
630 X=PEEK(I):I=I+1:P$=P$+C$(X)
640 IF L(X)=0 THEN 780
650 '
660 ' GET SECOND BYTE
670 A=PEEK(I):I=I+1
680 GOSUB 210:Y$=A$
690 IF L(X)=2 THEN 730
700 P$=P$+"BY"+Y$
710 GOTO 780
720 '
730 ' GET THIRD BYTE
740 A=PEEK(I):I=I+1
750 GOSUB 210:P$=P$+"H"+A$+Y$
760 '
770 ' PRINT THE LINE
780 PRINT P$
790 '
800 ' EXTRA LINE FOR JMP OR RET?
810 IF X<>195 AND X<>201 THEN 860
820 P$=";"
830 PRINT P$
840 '
```

```
850 ' LOOP BACK
860 IF I>I2 THEN STOP
870 GOTO 580
880 '
890 ' THE DATA SECTION
900 DATA "NOP",     "LXI B,"
910 DATA "STAX B", "INX B"
920 DATA "INR B",  "DCR B"
930 DATA "MVI B,", "RLC"
940 DATA "-",       "DAD B"
950 DATA "LDAX B", "DCX B"
960 DATA "INR C",  "DCR C"
970 DATA "MVI C,", "RRC"
980 DATA "-",       "LXI D,"
990 DATA "STAX D", "INX D"
1000 DATA "INR D",  "DCR D"
1010 DATA "MVI D,", "RAL"
1020 DATA "-",       "DAD D"
1030 DATA "LDAX D", "DCX D"
1040 DATA "INR E",  "DCR E"
1050 DATA "MVI E,", "RAR"
1060 DATA "RIM",     "LXI H,"
1070 DATA "SHLD ",  "INX H"
1080 DATA "INR H",  "DCR H"
1090 DATA "MVI H,", "DAA"
1100 DATA "-",       "DAD H"
1110 DATA "LHLD ",  "DCX H"
1120 DATA "INR L",  "DCR L"
1130 DATA "MVI L,", "CMA"
1140 DATA "SIM",     "LXI SP,"
1150 DATA "STA ",   "INX SP"
1160 DATA "INR M",  "DCR M"
1170 DATA "MVI M,", "STC"
1180 DATA "-",       "DAD SP"
1190 DATA "LDA ",   "DCX SP"
1200 DATA "INR A",  "DCR A"
1210 DATA "MVI A,", "CMC"
1220 DATA "MOV B,B","MOV B,C"
1230 DATA "MOV B,D","MOV B,E"
1240 DATA "MOV B,H","MOV B,L"
1250 DATA "MOV B,M","MOV B,A"
1260 DATA "MOV C,B","MOV C,C"
1270 DATA "MOV C,D","MOV C,E"
1280 DATA "MOV C,H","MOV C,L"
1290 DATA "MOV C,M","MOV C,A"
1300 DATA "MOV D,B","MOV D,C"
1310 DATA "MOV D,D","MOV D,E"
1320 DATA "MOV D,H","MOV D,L"
1330 DATA "MOV D,M","MOV D,A"
```

```
1340 DATA "MOV E,B","MOV E,C"
1350 DATA "MOV E,D","MOV E,E"
1360 DATA "MOV E,H","MOV E,L"
1370 DATA "MOV E,M","MOV E,A"
1380 DATA "MOV H,B","MOV H,C"
1390 DATA "MOV H,D","MOV H,E"
1400 DATA "MOV H,H","MOV H,L"
1410 DATA "MOV H,M","MOV H,A"
1420 DATA "MOV L,B","MOV L,C"
1430 DATA "MOV L,D","MOV L,E"
1440 DATA "MOV L,H","MOV L,L"
1450 DATA "MOV L,M","MOV L,A"
1460 DATA "MOV M,B","MOV M,C"
1470 DATA "MOV M,D","MOV M,E"
1480 DATA "MOV M,H","MOV M,L"
1490 DATA "HLT",    "MOV M,A"
1500 DATA "MOV A,B","MOV A,C"
1510 DATA "MOV A,D","MOV A,E"
1520 DATA "MOV A,H","MOV A,L"
1530 DATA "MOV A,M","MOV A,A"
1540 DATA "ADD B",  "ADD C"
1550 DATA "ADD D",  "ADD E"
1560 DATA "ADD H",  "ADD L"
1570 DATA "ADD M",  "ADD A"
1580 DATA "ADC B",  "ADC C"
1590 DATA "ADC D",  "ADC E"
1600 DATA "ADC H",  "ADC L"
1610 DATA "ADC M",  "ADC A"
1620 DATA "SUB B",  "SUB C"
1630 DATA "SUB D",  "SUB E"
1640 DATA "SUB H",  "SUB L"
1650 DATA "SUB M",  "SUB A"
1660 DATA "SBB B",  "SBB C"
1670 DATA "SBB D",  "SBB E"
1680 DATA "SBB H",  "SBB L"
1690 DATA "SBB M",  "SBB A"
1700 DATA "ANA B",  "ANA C"
1710 DATA "ANA D",  "ANA E"
1720 DATA "ANA H",  "ANA L"
1730 DATA "ANA M",  "ANA A"
1740 DATA "XRA B",  "XRA C"
1750 DATA "XRA D",  "XRA E"
1760 DATA "XRA H",  "XRA L"
1770 DATA "XRA M",  "XRA A"
1780 DATA "ORA B",  "ORA C"
1790 DATA "ORA D",  "ORA E"
1800 DATA "ORA H",  "ORA L"
1810 DATA "ORA M",  "ORA A"
1820 DATA "CMP B",  "CMP C"
```

```
1830 DATA "CMP D",   "CMP E"
1840 DATA "CMP H",   "CMP L"
1850 DATA "CMP M",   "CMP A"
1860 DATA "RNZ",     "POP B"
1870 DATA "JNZ ",    "JMP "
1880 DATA "CNZ ",    "PUSH B"
1890 DATA "ADI ",    "RST 0"
1900 DATA "RZ",      "RET"
1910 DATA "JZ ",     "-"
1920 DATA "CZ ",     "CALL "
1930 DATA "ACI ",    "RST 1"
1940 DATA "RNC",     "POP D"
1950 DATA "JNC ",    "OUT "
1960 DATA "CNC ",    "PUSH D"
1970 DATA "SUI ",    "RST 2"
1980 DATA "RC",      "-"
1990 DATA "JC ",     "IN "
2000 DATA "CC ",     "-"
2010 DATA "SBI ",    "RST 3"
2020 DATA "RPO",     "POP H"
2030 DATA "JPO ",    "XTHL"
2040 DATA "CPO ",    "PUSH H"
2050 DATA "ANI ",    "RST 4"
2060 DATA "RPE",     "PCHL"
2070 DATA "JPE ",    "XCHG"
2080 DATA "CPE ",    "-"
2090 DATA "XRI ",    "RST 5"
2100 DATA "RP",      "POP PSW"
2110 DATA "JP ",     "DI"
2120 DATA "CP ",     "PUSH PSW"
2130 DATA "ORI ",    "RST 6"
2140 DATA "RM",      "SPHL"
2150 DATA "JM ",     "EI"
2160 DATA "CM",      "-"
2170 DATA "CPI ",    "RST 7"
2180 DATA 0,2,0,0
2190 DATA 0,0,1,0
2200 DATA 0,0,0,0
2210 DATA 0,0,1,0
2220 DATA 0,2,0,0
2230 DATA 0,0,1,0
2240 DATA 0,0,0,0
2250 DATA 0,0,1,0
2260 DATA 0,2,2,0
2270 DATA 0,0,1,0
2280 DATA 0,0,2,0
2290 DATA 0,0,1,0
2300 DATA 0,2,2,0
```

```
2310 DATA 0,0,1,0
2320 DATA 0,0,2,0
2330 DATA 0,0,1,0
2340 DATA 0,0,0,0
2350 DATA 0,0,0,0
2360 DATA 0,0,0,0
2370 DATA 0,0,0,0
2380 DATA 0,0,0,0
2390 DATA 0,0,0,0
2400 DATA 0,0,0,0
2410 DATA 0,0,0,0
2420 DATA 0,0,0,0
2430 DATA 0,0,0,0
2440 DATA 0,0,0,0
2450 DATA 0,0,0,0
2460 DATA 0,0,0,0
2470 DATA 0,0,0,0
2480 DATA 0,0,0,0
2490 DATA 0,0,0,0
2500 DATA 0,0,0,0
2510 DATA 0,0,0,0
2520 DATA 0,0,0,0
2530 DATA 0,0,0,0
2540 DATA 0,0,0,0
2550 DATA 0,0,0,0
2560 DATA 0,0,0,0
2570 DATA 0,0,0,0
2580 DATA 0,0,0,0
2590 DATA 0,0,0,0
2600 DATA 0,0,0,0
2610 DATA 0,0,0,0
2620 DATA 0,0,0,0
2630 DATA 0,0,0,0
2640 DATA 0,0,0,0
2650 DATA 0,0,0,0
2660 DATA 0,0,2,2
2670 DATA 2,0,1,0
2680 DATA 0,0,2,0
2690 DATA 2,2,1,0
2700 DATA 0,0,2,1
2710 DATA 2,0,1,0
2720 DATA 0,0,2,1
2730 DATA 2,0,1,0
2740 DATA 0,0,2,0
2750 DATA 2,0,1,0
2760 DATA 0,0,2,0
2770 DATA 2,0,1,0
2780 DATA 0,0,2,0
```

```
2790 DATA 2,0,1,0
2800 DATA 0,0,2,0
2810 DATA 2,0,1,0
2820 '
2830 END
```

Let's look at the program in detail. As you can see, most of the listing consists of DATA statements. Lines 900-2170 contain the mnemonics for all the CPU instructions, and lines 2180-2810 contain the number of additional bytes required for each instruction. These are the bytes needed for associated data or address information.

The first part of the program consists mainly of subroutines for converting numbers between decimal and hexadecimal notation. The main part of the program begins on line 470 and extends to line 870. The first few lines of the main part of the program dimension and load the arrays C$ and L, which hold the mnemonics and the instruction lengths as stored in the DATA statements. The next few lines (530-560) input the starting and ending addresses for the disassembly. The subroutine at line 380 is called to convert from hexadecimal to the normal decimal internal form for numbers. Upon return, the starting value is stored in the variables I1 and I, and the ending value is stored in the variable I2.

Line 580 marks the top of the main loop. Here the current address in the variable I is converted into a hexadecimal value stored in the string I$. On line 600, the string P$ is defined to contain the beginning of the output line. You can see that the output begins with the address of the instruction, prefixed by an "H". This makes each address into a label for the line of assembly language. We used "H" to stand for hexadecimal address.

Next, the first byte of instruction is fetched into the variable X and looked up in the lists C$ and L. The mnemonic in C$ is added to the output string P$. If the instruction requires only one byte, then the program jumps to the end of the loop, where the line is printed out.

If there is a second byte, it is picked up. If there is no third byte, the second byte is prefixed by a "BY" and added to the output string P$, and then the program jumps to the end of the loop, where P$ is printed. We used "BY" to indicate that the data is stored in a byte.

If there is a third byte, it is prefixed with an "H", packed with the second byte, and added to P$. This makes a label operand, perhaps matching one of the "H" labels at the beginning of one of the other output lines of this program. At the bottom of the loop, P$ is printed out in this case as well.

At the very bottom of the main loop, a check is made for JuMP or RETurn instructions. Following these, we place an extra blank comment line to make the code more readable.

This disassembler program can be used to examine each routine that we discuss in this book and to discover other routines as well. It can be modified in a number of ways. For example, it can be changed to produce a list of addresses referenced by the machine code together with the addresses of the instructions where those references occur. These references can be dumped into a file and sorted, giving a list of cross-references. From this list you can find the key entry points and variables of the machine-code programs.

Other BASIC programs can be written using the same hexadecimal conversion subroutines. For example, it is easy to write a program that displays the memory as a list of hexadecimal 16-bit words. This is useful for displaying tables of addresses. For example, such a table containing the addresses of the routines to handle all the BASIC commands starts at location 80h = 128d.

Searching for Special Patterns

From time to time it is important to find certain patterns in memory. For example, you will see on pages 4-7 that output ports F0h = 240d through FFh = 255d are used for the Liquid Crystal Display screen. Hence the key instruction for control of the LCD is:

```
OUT  port
```

where port is a number from F0h to FFh (240 to 255 decimal).

Let's write a program that searches for all instances of this instruction. The machine code for the OUT instruction is 211 decimal. Here is a short program that prints out the address of each occurrence of the pattern: 211, x, where x is a decimal number between 240 and 255.

```
100 ' PATTERN SEARCH
110 '
120   FOR I = 0 TO 32767
130    X = PEEK(I)
140    IF X<>211 THEN 180
150    Y = PEEK(I+1)
160    IF Y<240 THEN 180
170    PRINT I;
180  NEXT I
```

The program consists of a FOR loop that ranges through the entire ROM for address I from 0 to 32767 (see line 120). On line 130, the value of the byte is fetched, and on line 140 it is checked to see if it is equal to 211, the code for the OUT instruction. The next part of the FOR loop is skipped if a match is not made. If the match is made, the next byte is checked. If it is not between 240 and 255, the next part of the FOR loop is skipped. If this second byte value is between 240 and 255, the address I is printed out and the loop continues.

When you run this program, you will discover that there are six locations where this sequence occurs. In Chapter 4, when we study the LCD in more detail, we will discuss how the machine language at these locations actually programs the LCD.

This program can be easily modified to look for other patterns. For example, you could change it to look for specified three-byte sequences.

Notes on Using This Book

Chapters 1 and 2 of this book are introductory, providing you with an overview, guidance, and tools that you will need for reading and making use of the rest of the book.

Each of the remaining chapters (3 through 9) covers a major area of the operation of the Model 100. These chapters start out with a general discussion and then delve into a particular set of ROM routines.

The Boxes

Key information about major ROM routines is displayed in boxes. Each box shows the vital statistics for one particular routine or block of code.

Each box begins with the *name* of the routine. Some routines have a single word in capital letters as their name. These are the routines that are described in documents and articles published by Radio Shack (see Radio Shack publication 700-2245, *Model 100 ROM Functions*). Other routines have names consisting of several words. These are not mentioned in Radio Shack's literature.

After the name comes the *purpose* of the routine. This is a short description of the major function that the routine performs.

Following the purpose is the *entry point* for the routine. This is the address where execution of the routine begins. The entry address is given in both hexadecimal and decimal form, as described below.

The *input* and *output* for the routine are shown next. These describe how the registers and memory are used to pass data in and out of the routine. Here we also often mention the physical I/O devices that are involved.

Next, a BASIC example is given, if applicable. Normally this is in the form of a CALL command. The BASIC CALL command allows you to call a machine-language routine, passing values to it via the A register and the HL register pair. The CALL command does not allow you to read directly into your program any data that might be stored in a register, such as the A register. Because of these conditions, we have given BASIC examples just for those routines that input data through only the A and HL registers and do not output data through CPU registers.

Each box ends with a spot for *special comments*. Here we place general comments or warnings about using the routine.

Address Formats

As noted above, memory addresses are shown in both hexadecimal and decimal format. For example, 646h = 1606d. Here the small letter "h" following the 646 indicates a hexadecimal number, and the small "d" following 1606 indicates a decimal number. Entry points and other addresses are thus accessible both to the BASIC programmer, who needs addresses in decimal, and to the assembly-language programmer, who normally works in hexadecimal (as when using the disassembler).

Although some BASICs have built-in functions to use both decimal and hexadecimal forms of numbers, the Model 100's BASIC can work only with decimal notation. We have also provided simple routines for converting between decimal and hexadecimal, but you would not want to include these routines in every program that CALLed a ROM routine.

BASIC Programs

In this book there are more than two dozen BASIC programs that you can type in and run on the Model 100. You can store these programs as RAM files and/or as cassette files on tape. Many people store such programs by uploading them to a larger system, where they are saved on disk, or you can use the optional Model 100 disk. The programs fall into three main classes: *tools*, *demonstrations*, and *applications*.

Programs in the first category, *tools*, are presented in Chapter 1. Here we have programs that dump, disassemble, and search memory.

The second category, *demonstrations*, is designed to give you a better feel for the way particular features work. Sometimes we display key memory locations that are affected by a process. Sometimes we show the direct results of a software action upon some hardware. By running and modifying these programs yourself, you should gain the deeper understanding that is possible only through "hands on" experience.

The third category, *applications*, somewhat overlaps the last category. Applications programs demonstrate features and give useful and interesting applications for machine language and machine-level features of the Model 100. For example, we have included programs that interactively interpret formulas, scroll selected lines of the LCD display screen, and automatically dial a telephone number.

Summary

In this chapter we have discussed how this book is organized and how to use it. We have also presented some basic program tools that can be used in conjunction with this book. With these tools, you can better understand this book and also extend your knowledge to areas beyond the scope of these pages.

2

The Model 100's Hardware

Concepts

- 80C85 central processor
- Bus structure
- ROM and RAM
- Port decoding
- 8155 parallel input/output interface controller
- μPD 1990AC real-time clock
- 6402 universal asynchronous receiver transmitter
- LCD screen
- Keyboard
- Printer interface

In this chapter we'll explore the organization of the Model 100's hardware. We will begin with an overview of the entire internal structure and then survey each of the major subsystems. Some of these subsystems are represented by chips such as the 80C85 CPU, the 8155 PIO, the μPD 1990 clock, and the 6402 UART. Other systems such as the memory, LCD, keyboard, and printer involve combinations of chips and other components. We will also study the buses that connect all of these subsystems together physically and logically to form the Model 100 computer.

Why Study Hardware?

Before we plunge into the hardware, let's talk about why it is useful for a programmer to know about such things. The answer is simple: the hard-

ware is the stage upon which the software plays. The hardware physically houses the software and gives it meaning. More explicitly, the machine-language instructions of the CPU are part of the hardware, and without them, the programs stored in the ROM and RAM are merely sequences of numbers. This notion extends beyond the CPU. Each of the other major chips, including the 8155 PIO, the μPD 1990 clock, and the 6402 UART, can be programmed according to certain rules that involve sending bytes out certain I/O ports. Without these rules, these byte sequences are meaningless. To understand these rules, you need an understanding of the structure of the machine and its various subsystems. Thus, to understand the software, at least at the machine-language level, you must understand the hardware.

However, you should not take this to mean that you need to understand every detail in this chapter in order to profit from the rest of the book. Much of the material in this chapter will be relevant only in certain programming situations. Thus, while you are reading this chapter you should not worry if some aspects of the operation of the hardware are not completely clear. The detailed descriptions of the ROM routines in future chapters will clarify many points.

Each chip in the Model 100 is fully described in the documentation published by the company that designed it. For the programmable chips, this includes all their programming rules. Much of this information is also included in the service manual for the Model 100. We will present the essential details of this material in this chapter.

Hardware Overview

We start with a map of the Model 100 hardware system (see Figure 2-1).

The CPU is the nerve center of the system. It controls the main bus, which runs through the system like a spinal column, linking the various subsystems to the CPU. Two devices, the bar code reader and the tape cassette recorder/player interface, connect directly to the CPU.

The main system bus controls the serial communications lines through the 6402 UART chip, the RAM and ROM, and the 8155 PIO chip. It also helps to control and monitor the keyboard, clock, and LCD display. The main system bus is terminated in a socket in a recessed area on the bottom of the Model 100. A socket for the optional ROM is also here. This socket can be used for other purposes as well, such as connecting the disk drive and video interface.

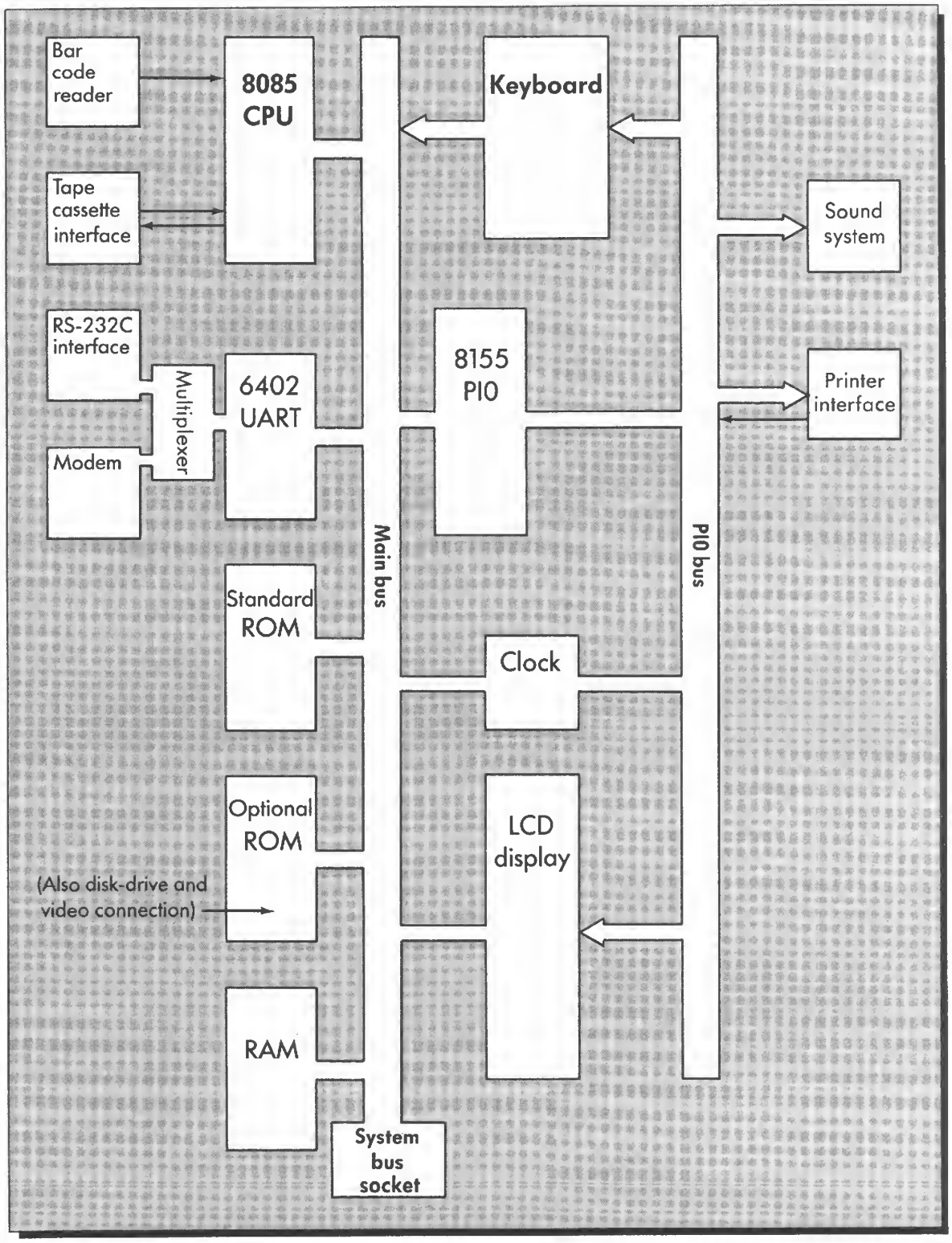


Figure 2-1. Model 100 block diagram

The 8155 PIO is the second main chip in the system. It contains a timer and controls a second bus that we call the PIO bus. The PIO bus feeds multiple signals in parallel to the keyboard, printer interface, and LCD display. It feeds other signals to the sound system and clock. Thus, control of all of these devices is channeled through the 8155 PIO.

The 80C85 Central Processor

Now let's study the individual subsystems, starting with the CPU.

The Model 100 uses a CMOS version of the popular 8-bit 8085 central processor chip (hence the designation 80C85). To a programmer the CMOS version of this chip behaves identically to the regular version. The only real difference is in power consumption. Because of its low power consumption, the CMOS version is able to run on battery power for many hours.

As a rule, low-power devices run slower than their high-power counterparts. However, much progress is being made in this regard. The CMOS 8085 in the Model 100 is driven at a clock speed of 2,457,600 cycles per second. Although this is not as fast as most of the high-power versions of the current crop of processors, it is significantly faster than the first versions of its earlier cousin chip, the 8080. As a result, the Model 100 runs faster than many of the larger microcomputers did a few years ago.

The 8085 chip differs from the older 8080 CPU chip in several important ways, including its RIM and SIM instructions, additional interrupts, internal clock circuits, and simpler power requirements. Except for the RIM and SIM instructions, the 8085 uses the same machine language and the same assembly-language mnemonics as the 8080. This overlap means that there is a large base of information, software, and expertise available for the 8085 CPU.

We are assuming that you already have access to books and manuals on the 8080 or 8085. Therefore, this book will not present a detailed discussion of these 8-bit microprocessor chips and how to program them.

Figure 2-2 shows the 8085 microprocessor's internal structure. We will discuss the major features of this diagram.

The internal structure of the 8085 CPU consists of eight major parts: CPU registers, Arithmetic Logic Unit (ALU), timing and control, interrupt control, serial control, bus interfacing, and instruction decoding.

The CPU registers each have their own "personalities". We assume that you have already been introduced to and understand these personalities from prior experience. Here is a quick overview of what you should know. Registers A, B, C, D, E, H, L, and Flags are all 8-bit memory cells within the CPU that can be combined into the following 16-bit register pairs: A/

Flags, BC, DE, and HL. Some CPU instructions work with registers as single 8-bit temporary memory cells, while others work with these register pairs as 16-bit temporary memory cells. The A register is used as an "accumulator" for data; the HL register pair is used as a "pointer" into memory; the SP is the Stack Pointer for pointing to the stack for storing data and return addresses; and the PC is the Program Counter for pointing to the individual bytes of the machine language for the CPU.

The Arithmetic Logic Unit (ALU) performs the 8-bit arithmetic and logic operations such as addition, subtraction, complement, AND, and OR. It also performs shifting operations.

The timing and control section keeps the 8085 CPU humming along and also provides timing and control signals for the rest of the computer. The basic timing frequency is derived from an external crystal. The input from the external crystal comes into the timing and control section, where

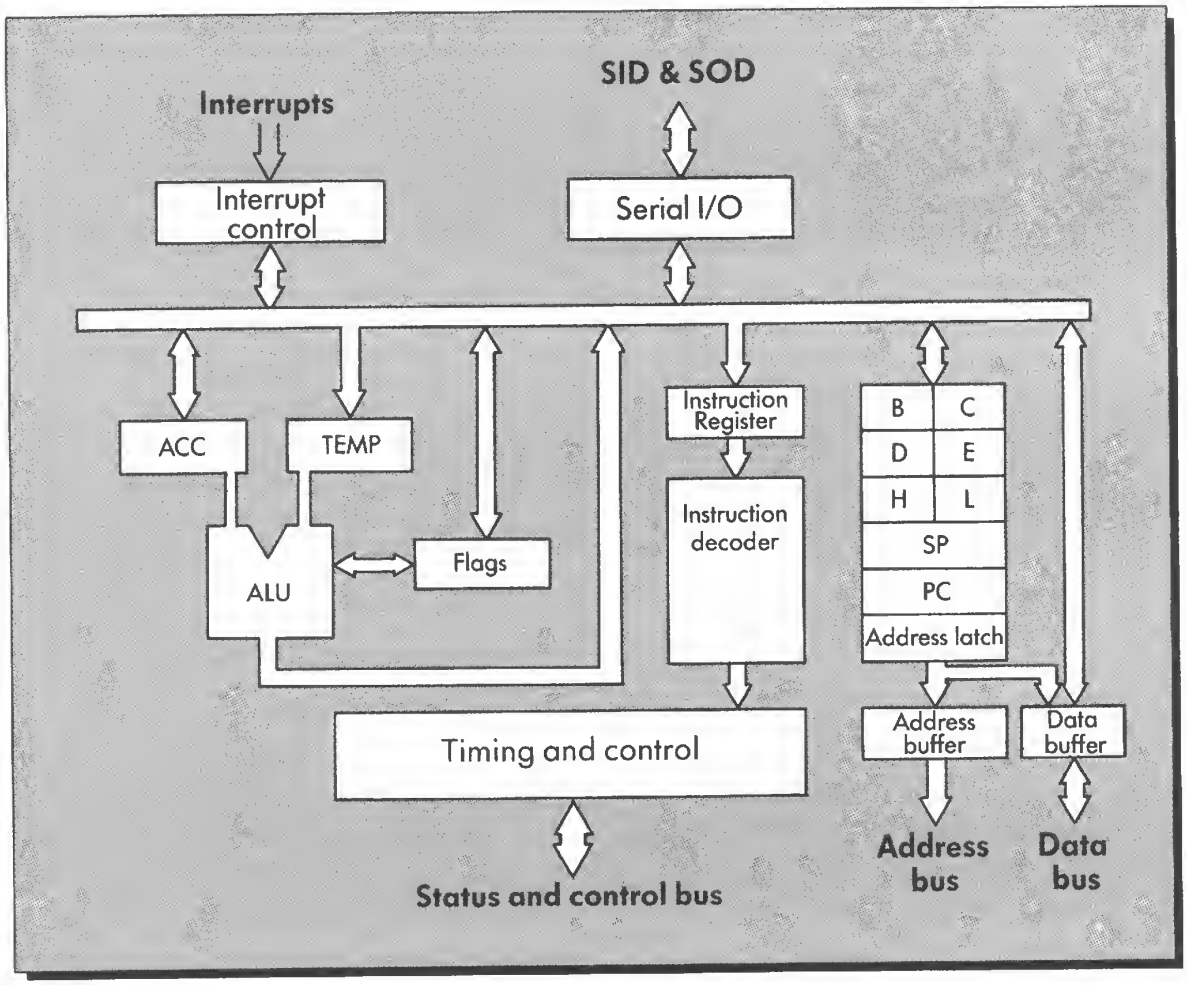


Figure 2-2. 8085 microprocessor (internal structure)

it is used to derive the basic timing signal for the operation of the microprocessor chip and the rest of the computer. For the Model 100, the crystal oscillates at 4,915,200 cycles per second. The timing and control circuits divide the crystal frequency by 2 to give the clock signal for the system. The same signal is sent out of the CPU to control memory and I/O access.

The interrupt control section interfaces the CPU to interrupt signal lines coming from various devices in the computer. These interrupt lines are called TRAP, RST 5.5, RST 6.5, and RST 7.5. In Chapter 3 we will see how these lines are serviced by software routines. For now, you need to understand only that if interrupts are enabled, a pulse on the TRAP line causes the CPU to automatically branch to location 24h = 36d as though a CALL 24h were suddenly executed. The RST 5.5, RST 6.5, and RST 7.5 lines operate in a similar manner, branching to locations 2Ch = 44d, 34h = 52d, and 3Ch = 60d respectively. The three RST interrupts behave somewhat differently from the TRAP interrupt, however, in that the 8085 RIM and SIM instructions can selectively enable and disable them.

The serial control section connects the Serial Output Data (SOD) and Serial Input Data (SID) pins to the CPU. These are serviced by the RIM and SIM instructions. In Chapter 9 we will see how the cassette interface uses these instructions to send its data in and out of the SOD and SID pins.

The bus interfacing section connects the main internal data and address buses with the external data and address buses. Internally the 8085 CPU has separate buses for addresses and data, but externally the data and the lower eight bits are shared by the same eight pins coming out of the 8085 chip. Some "buffering" (temporary storage) and "multiplexing" (switching) are done by the interfacing section.

The instruction decoding section is responsible for reading machine code instructions as they come into the computer byte by byte from memory. As they are being decoded, these bytes are stored in the Instruction Register (IR). The instruction decoding section analyzes these bytes, using the various bit fields to determine what the CPU is to do, what data it is to use, and where it is to put the results.

The System's Buses

The Model 100 has a bus system that includes power, control, data, and address subbuses. Since most of the computer is on one board, the bus structure is not absolutely well defined. However, there is a bus extension socket that brings certain signals out of the computer in a very well-defined manner (see Figure 2-3).

Power

The power bus carries the power needed to run the electrical components in the system. There are four separate lines that help conduct power: a ground line (GND) at 0 volts, a +5 volt line (VDD), a −5 volt line (VEE), and a line that carries voltage from the ni-cad battery (VB).

The voltage for the VDD and VEE lines is supplied by the four AA batteries or the AC adapter if it is used. The regular power switch on the right side of the Model 100 controls the power to these lines. The VDD line supplies the power for most of the chips in the Model 100. The VDD and VEE lines work together in certain circuits to produce voltages greater than five volts as needed. For example, the RS-232, MODEM, and LCD circuits use both the VDD and VEE lines.

As mentioned before, the voltage for the VB line is supplied by the rechargeable ni-cad battery. The memory power switch on the bottom of the Model 100 controls the power to this line. The VB line supplies power to the Model 100's RAM chips and to the real-time clock chip. It keeps the clock running and allows the contents of the RAM to be retained when the main power is shut off.

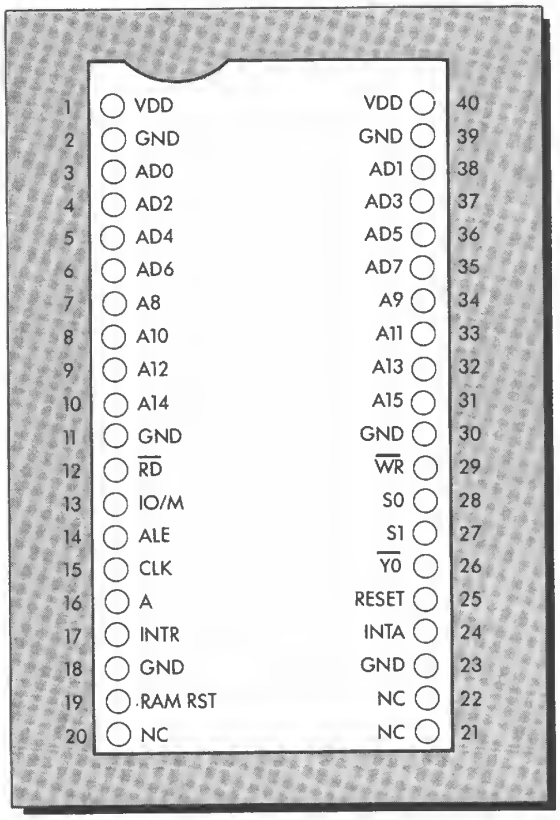


Figure 2-3. Bus extension socket

Of these power lines, only the ground and VDD lines appear on the bus extension socket.

Control

The control subbus contains a variety of signal lines, including read and write command lines for memory and I/O, interrupt control, timing signals, status, and reset control. We will not discuss these lines in much detail, since they are not of great concern to a programmer as long as they do their job properly.

Of the system's control lines, the following appear on the extension socket: RD, IO/M, ALE, CLK, A, INTR, RAM RST, WR, S0, S1, Y0, RESET, and INTA. All but the A, RAM RST, and Y0 are standard signals of the 8085 CPU. The A signal indicates when either reading or writing is happening and is used by external RAM, if present. The RAM RST signal is used to enable and disable any such external RAM. The Y0 signal is used to select an optional I/O controller unit. Y0 is generated when ports 80h = 128d through 8Fh = 143d are selected.

Data

There are eight data lines in the data subbus. They are labeled D0 through D7. On the 8085 CPU, the data lines share the same pins with the lower eight address lines (they are therefore labeled ADO through AD7 on Figure 2-3). On the main circuit board, the data lines are separated from the address lines.

The eight data lines appear on the bus extension socket.

Address

There are sixteen address lines in the address subbus. They are labeled A0 through A15. These lines are separated from the data lines on the main circuit board. All sixteen address lines appear on the bus extension socket.

ROM and RAM

The ROM is implemented by one chip that is called by its model number: LH-535618. This chip has fifteen address lines, three control lines, and eight data lines.

The fifteen address lines are connected to the lower fifteen lines of the address bus. The sixteenth address line of the address bus is routed to one

of the control lines (Chip Select) of the ROM (see Figure 2-4). This ensures that the memory cells in this chip occupy the lower 32K bytes of the memory addressing space of the machine.

The data lines connect to the data bus, and the remaining two control lines control the timing of address selection and data output.

The RAM is implemented by four chip packs, each containing 8K bytes (see Figure 2-5). At least one of these chips must be installed for proper operation of the computer, for the various built-in ROM programs need some scratch storage. With just one chip pack installed, you have an 8K RAM machine. By installing more chips you can have a 16K, 24K, or 32K RAM machine. The chips are installed in high to low order in the addressing space of the 8085 CPU.

The RAM chip packs each contain four TC55188F-25 chips. Each of these chips has eight data lines, three control lines, and eleven address lines. This means that each chip contains two to the eleventh, or 2K, bytes of RAM. The Chip Enable control lines are used to determine which chip is

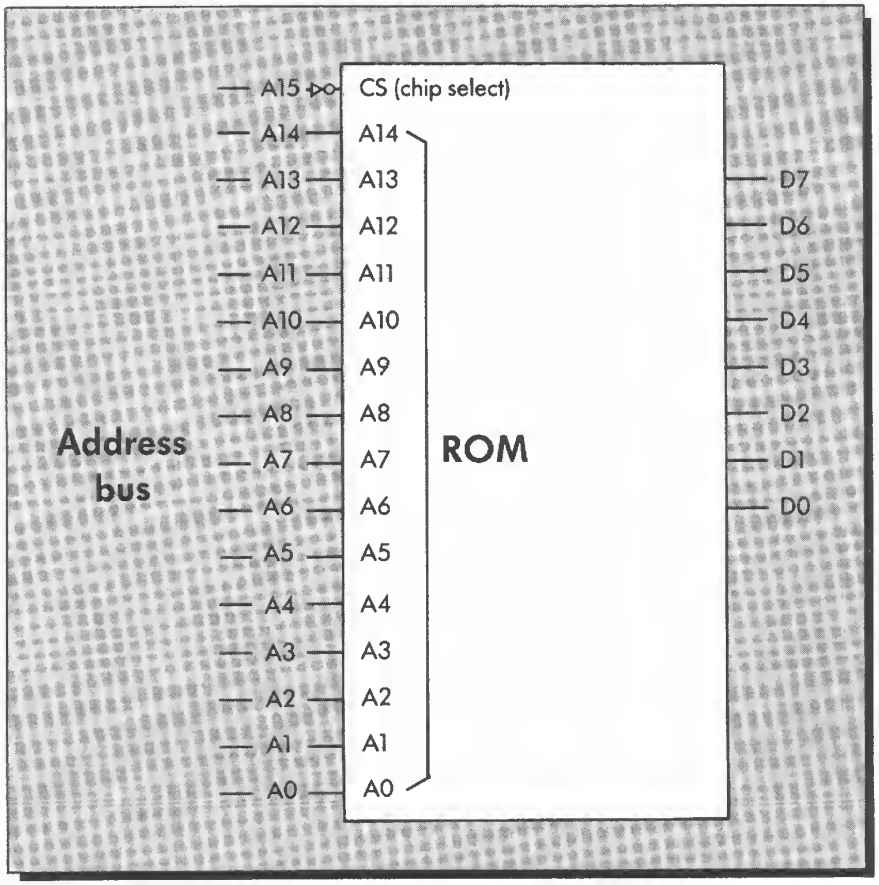


Figure 2-4. Addressing the ROM

selected for a particular memory address. Logic on the main circuit board converts address information from the address bus into the individual chip enable signals. In particular, bit 15 of the address bus determines whether or not the RAM is enabled at all, and bits 11 through 14 are decoded to determine which of the possible sixteen (four packs of four) chips is selected (see Figure 2-6).

Port Decoding

The 8085 CPU belongs to a family of processors that has two separate addressing spaces, one for memory and one for I/O. For the 8085 CPU, the memory addressing space has 64K bytes and the I/O addressing space has 256 bytes. In the former case, sixteen bits are used to generate addresses, and in the latter case only eight bits are used. For the I/O space, the lower eight lines of the address bus are used. A control signal in the control bus

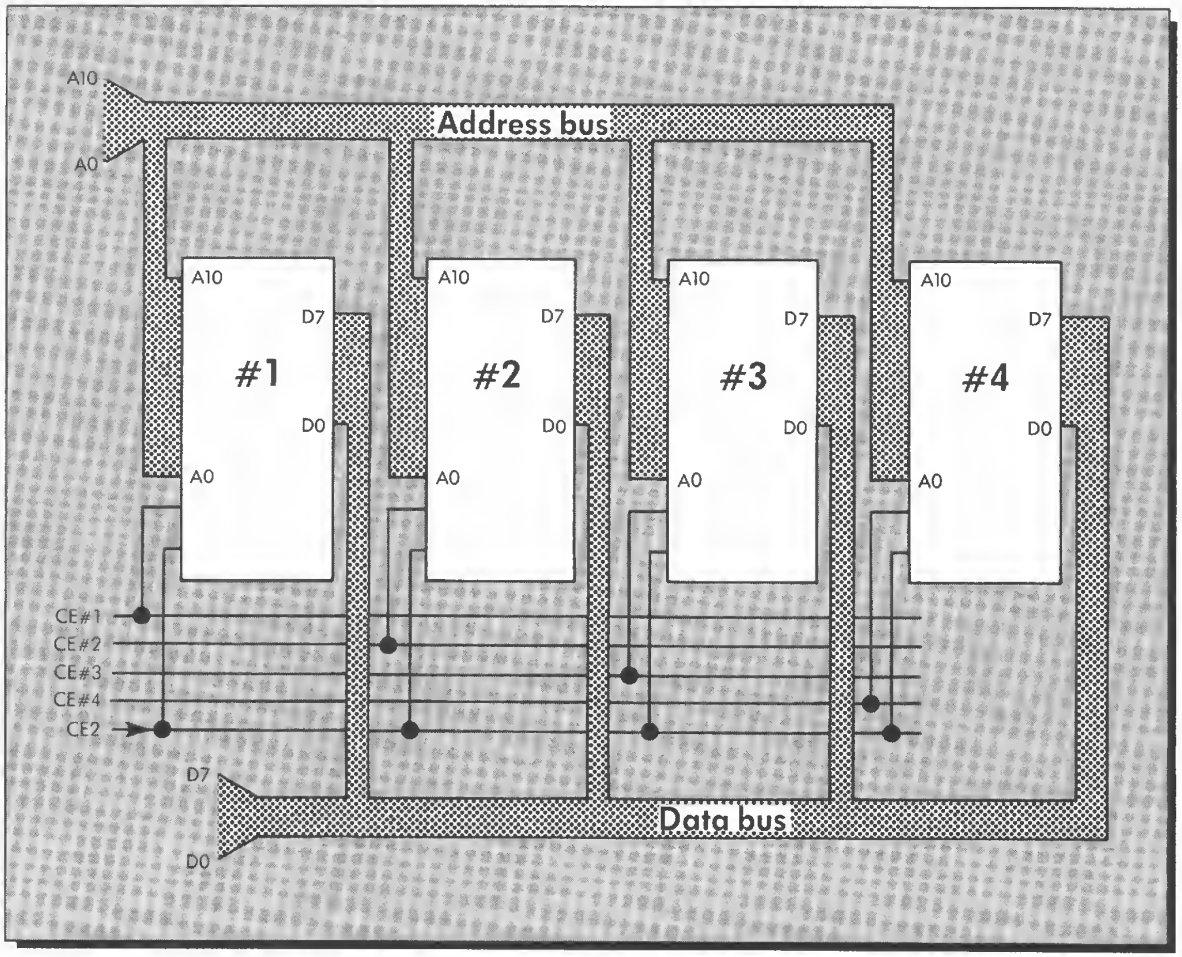


Figure 2-5. RAM chip pack

determines whether the CPU is trying to access a memory address or an I/O address. The programmer controls the CPU in this regard by selecting the appropriate machine-language instruction. IN and OUT instructions access the I/O space, whereas MOV, LDA, STA, LHLD, and SHLD instructions access the memory space.

The Model 100 uses a *decoder chip* to divide up the I/O addressing space into ranges for various devices (see Figure 2-7). This decoder chip takes the upper four bits of the eight-bit I/O address and produces eight individual control lines, Y0 through Y7. These control lines are then used to selectively enable the various devices in the Model 100.

Let's look at the decoding process in more detail. As this decoder chip is configured, it has four input lines (consisting of three selection lines and one enable line) and eight output lines (Y0 through Y7). If the enable line is zero, then all outputs are zero. In this case, we say that the decoder is disabled. However, if the enable line is equal to one, then the chip converts a three-bit selection code placed on the selection lines into a pattern of signals on the output lines, turning "on" the output line corresponding to

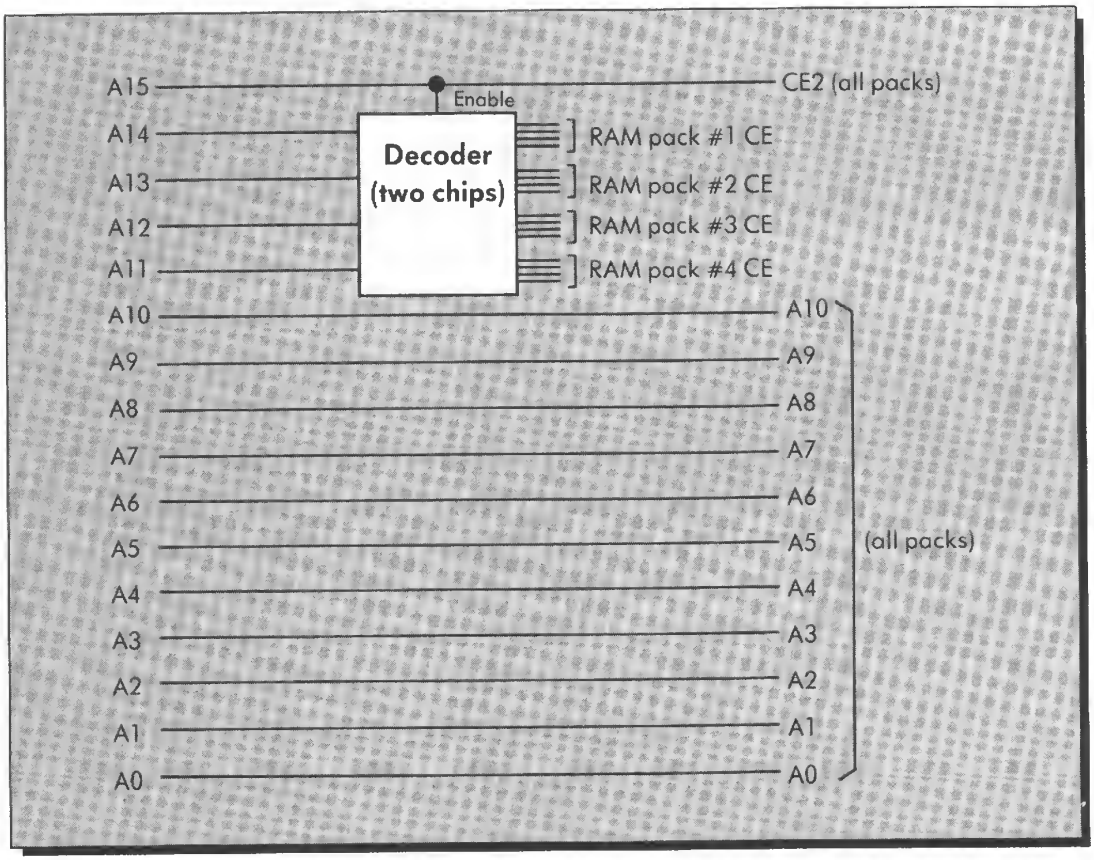


Figure 2-6. RAM addressing

the selection code and turning "off" all the rest. For example, if the three selection lines form the binary pattern 000, then Y0 is selected (has the value one) and the rest of the lines have value zero. If the three selection lines form the binary pattern 001, then Y1 is selected and the others are zero; and so on.

The port address lines A4, A5, and A6 are connected to the three selection lines, and the port address line A7 is connected to the enable line. Thus, if A7 is zero, all of the output lines Y0 through Y7 are zero; that is, none of the outputs is selected. This happens if the address is less than 80h = 128d.

If the address is greater than or equal to 80h = 128d, then the address line A7 has value one, enabling the decoder. In this case, the address bits A4, A5, and A6 determine which of the output lines Y0 through Y7 is selected.

The addresses from 80h to FFh divide into ranges depending upon the values for bit positions A4 through A6 of the address. For example, the range 80h = 128d through 8Fh = 143d corresponds to the pattern 000 for these bits, which selects Y0; the range 90h = 144d through 9Fh = 159d corresponds to the pattern 001, which selects Y1; the range A0h = 160d through AFh = 175d corresponds to the pattern 010, which selects Y2; and so on to the range F0h = 255d through FFh = 255d, which corresponds to the pattern 111 and thus selects Y7.

Table 2-1 shows each addressing range, the control line it selects, and the functions within the system that it enables.

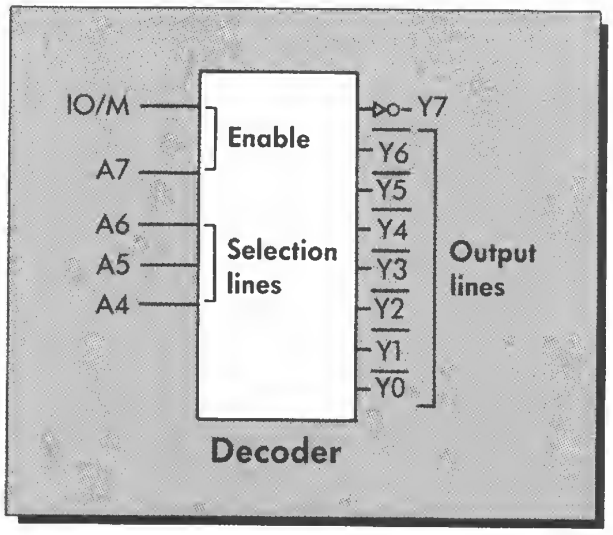


Figure 2-7. The port decoder

8155 Parallel Input/Output Interface Controller

The 8155 in the Model 100 is a CMOS version of the very powerful and flexible 8155 multifunction chip. This chip has 22 parallel I/O data lines, a timer, and 256 bytes of RAM (see Figure 2-8 for this chip's internal structure). The Model 100 uses all the data lines and the timer but does not use the RAM.

The 22 I/O data lines of the 8155 are divided into three I/O ports with eight lines in the first port (port A), eight in the second (port B), and six in the third (port C). We shall see in subsequent chapters how these ports are shared, in a very thrifty manner, by most of the input and output subsystems of the Model 100. In particular, the keyboard, real time clock, LCD screen, and printer all use some of these I/O data lines.

On the Model 100, the ports on the 8155 PIO chip are assigned the port addresses B0h = 176d through BFh = 191d. This is because the 8155 chip is enabled via the Y3 signal line from the main port decoder.

There are six ports used by the 8155 PIO. Each port appears twice within the sixteen-port range assigned to the 8155. This is because one bit (bit 3) of the port address is ignored.

Port 0 (at port addresses B0h = 176d and B8h = 184d) is used for control and status. When the port is written to, it is used for control. When it is read from, it is used for status.

When port 0 is used for control, bits 7 and 6 are used to set the timer mode as follows: 00 means no operation, 01 means stop the timer, 10 means stop after the count has expired, and 11 means start the timer. Bit 5 is used

Port Addressing Range	Functions
80h = 128d to 8Fh = 143d	Optional I/O controller
90h = 144d to 9Fh = 159d	Optional telephone answering unit
A0h = 160d to AFh = 175d	Output bit 0: on/off relay for telephone
	Output bit 1: enable Modem chip
B0h = 176d to BFh = 191d	8155 PIO
C0h = 192d to CFh = 207d	Data in/out for UART
D0h = 208d to DFh = 223d	UART program and read status
E0h = 224d to EFh = 239d	Output bit 0: select optional ROM
	Output bit 1: printer strobe
	Output bit 2: clock strobe
	Output bit 3: cassette motor on/off
	Input: keyboard matrix
F0h = 240d to FFh = 255d	LCD program, read, and write

Table 2-1. Port addressing ranges and functions

to enable or disable interrupts for port B; a value of 0 means disable. Bit 4 is used to enable or disable interrupts for port A. Bits 3 and 2 are used to define how the lines in port C are to be used as follows: 00 makes all six lines into input lines, 01 makes all six lines into output lines, 10 makes half the lines into output lines and half into control and status lines for port A, and 11 makes half the lines into control and status lines for port A and half the lines into control and status lines for port B. Bit 1 is used to control the direction of the data lines in port B, and bit 0 is used to control the direction of the data lines in port A. A value of 0 means that the lines are used for input, and a value of 1 means that the lines are used for output.

For the Model 100, bits 7 and 6 are changed to control the timer, but bits 5 through 0 always stay the same. These last six bits form a six-bit binary number, 000011b. This number specifies that all interrupts from ports A and B are disabled, ports A and B are used for output, and all six lines of port C are used for input. On the Model 100, ports A and B are

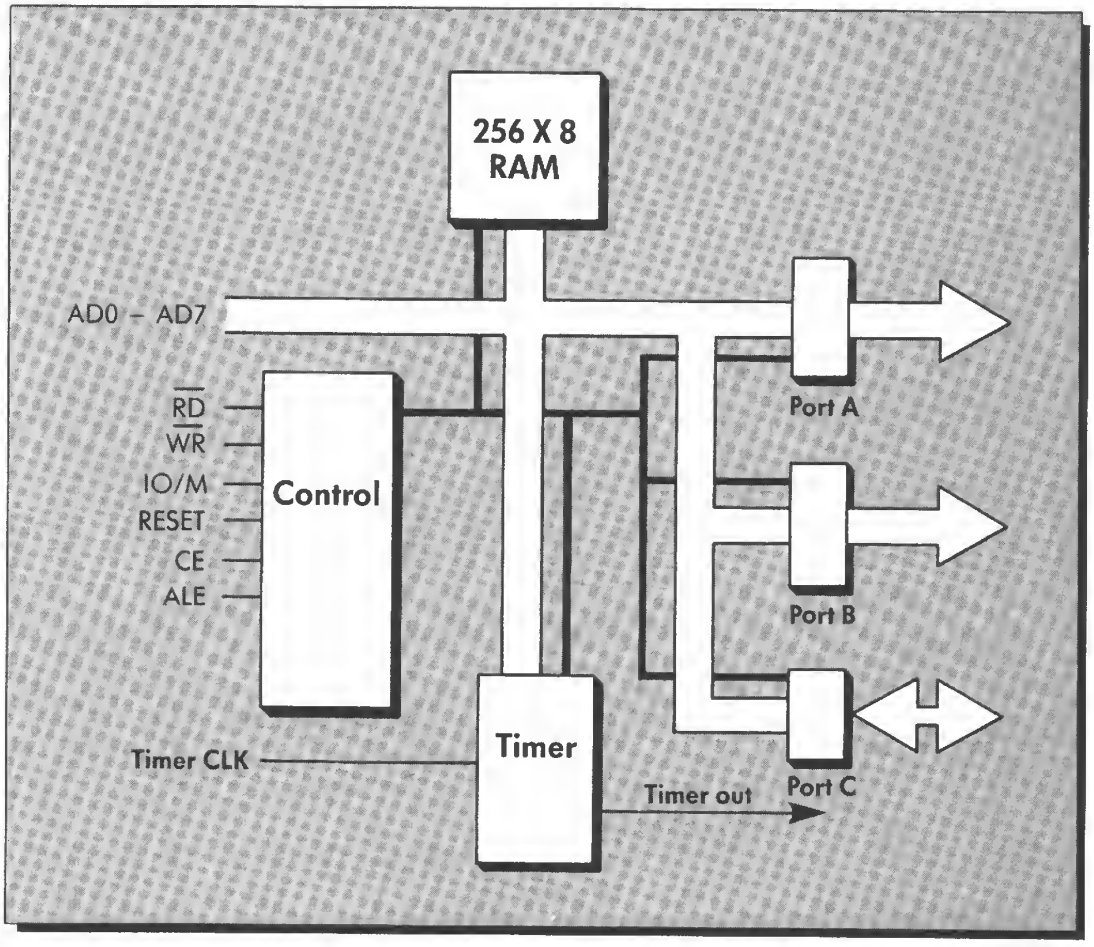


Figure 2-8. 8155 P10 (internal structure)

used to send control signals to various devices in the computer, while port C is used to monitor the status of various devices.

When port 0 is used for status, bit 7 is not used, bit 6 tells if the timer is running, and the other bits are used to monitor the interrupt and full/empty status of ports A and B. The Model 100 never reads port 0 (addresses B0h = 176d and B8h = 184d) and therefore never uses this status port.

Port 1 (addresses B1h = 177d and B9h = 185d) connects directly to the eight data lines of port A. Port 2 (addresses B2h = 178d and BAh = 186d) connects directly to the eight data lines of port B. Port 3 (addresses B3h = 179d and BBh = 187d) connects directly to port C.

Port 4 (addresses B4h = 180d and BCh = 188d) contains the lower eight bits of the count for initializing the timer. Port 5 (addresses B5h = 181d and BDh = 189d) contains the upper six bits of the count for the timer and two bits that are used to set the timer mode. See Chapter 8 on sound for more details on setting the timer.

Special Control Port

In addition to the control and status lines maintained by the 8155 PIO, there is a special control port that appears at addresses E0h = 224h through EFh = 239d (see Figure 2-9).

For this port, bit 0 is used to select an optional ROM, bit 1 is used to "strobe" data to the printer, bit 2 is used to "strobe" the real time clock, and bit 3 is used to control the remote motor control for the cassette recorder/player.

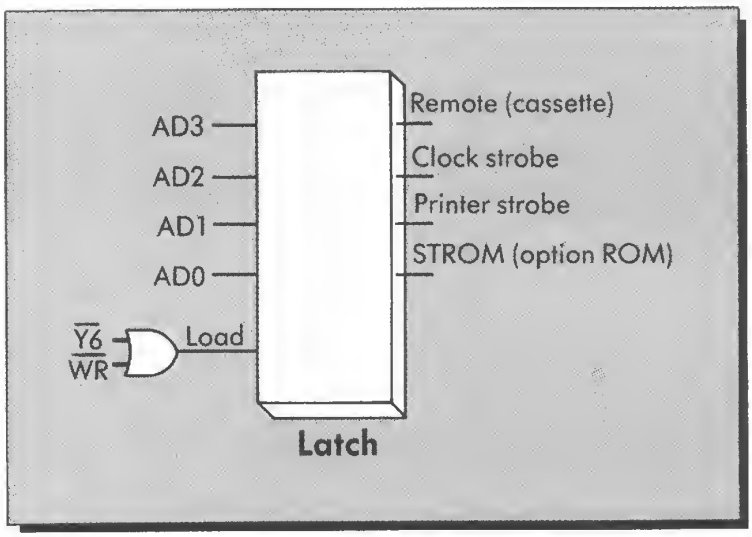


Figure 2-9. Special control port

The term "strobe" means that the signal line is used to send a pulse that locks the data being sent into the device to which it is being sent (printer, clock, or whatever). The strobe pulse is given once the data has been placed on the data lines and has settled down to a valid set of values.

μPD 1990AC Real Time Clock

The real time clock will be discussed in Chapter 5. We include its internal structure here (see Figure 2-10).

6402 Universal Asynchronous Receiver Transmitter

The 6402 UART (Universal Asynchronous Receiver Transmitter) will be discussed in Chapter 7. In that chapter we will see how the entire serial communications subsystem works. We include its internal structure here (see Figure 2-11).

Liquid Crystal Display Screen

The Liquid Crystal Display Screen will be discussed in Chapter 4. In

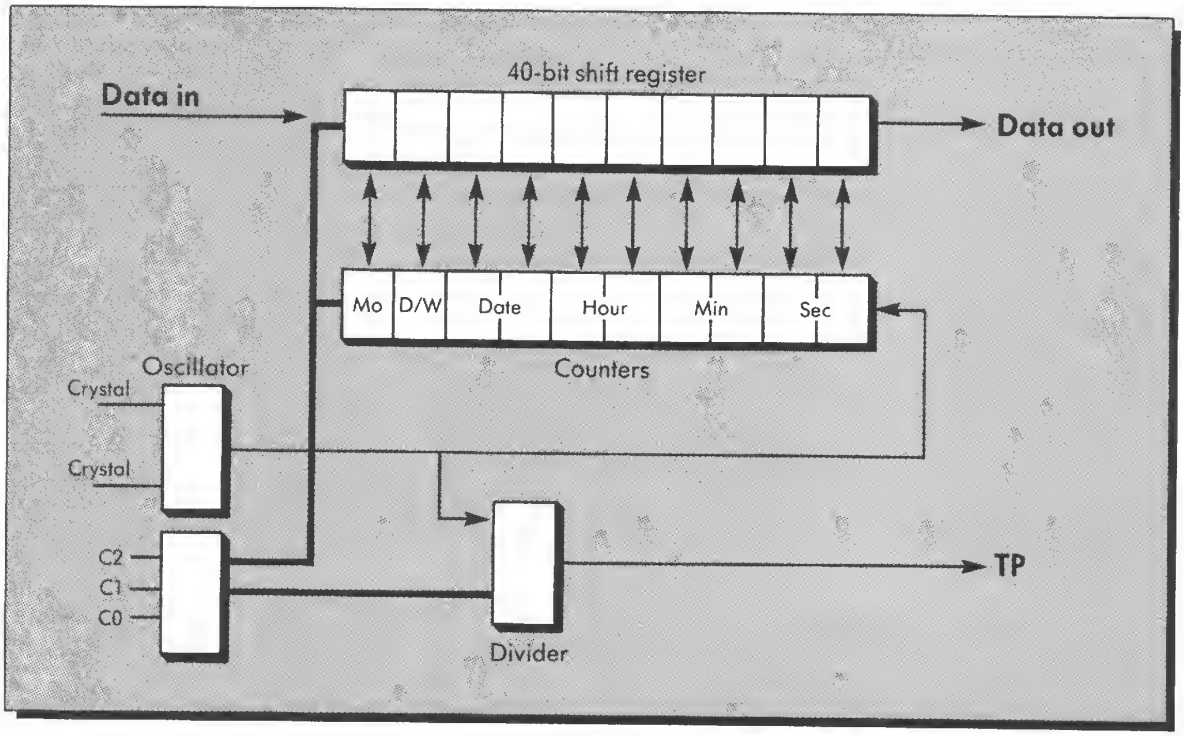


Figure 2-10. Real time clock (internal structure)

that chapter we will see how it works and how it connects to the 8155 I/O data lines and to the main address and data buses.

Keyboard

The keyboard will be discussed in Chapter 6. In that chapter we will see that the keys are laid out in a matrix that can be read using the data I/O lines from ports A and B of the 8155 and a special input port with addresses E0h = 224d through EFh = 239d.

Printer Interface

Let's finish this chapter with a quick discussion of the printer interface, since it will not be discussed later.

The printer interface uses port A of the 8155 PIO, the special control port (addresses E0h = 224h through EFh = 239d), and bits 1 and 2 of port C of the 8155 PIO chip (see Figure 2-12).

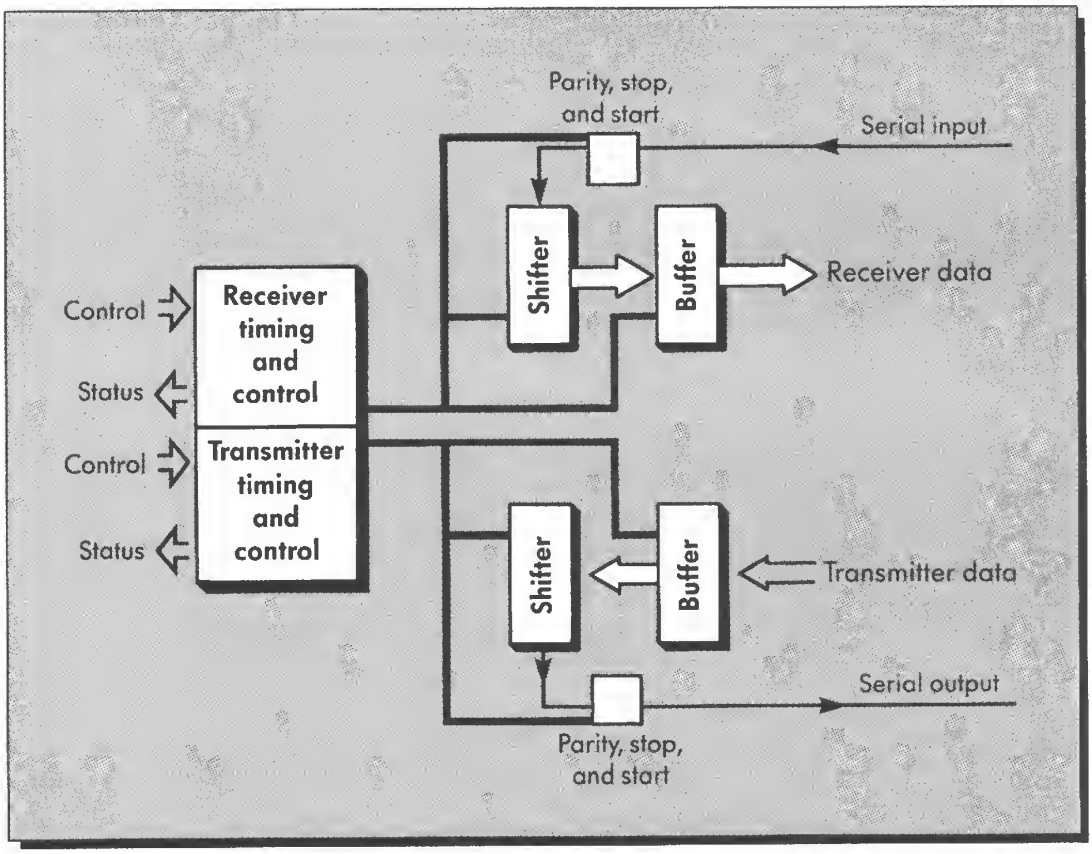


Figure 2-11. 6402 UART (internal structure)

Bits 1 and 2 of port C of the 8155 carry the busy/ready signal (noninverted and inverted) from the printer. To send data once the printer is ready, you send the data out port A of the 8155 and then change bit 1 of the special control port from a value of zero to a value of one and finally back to zero. This last action "strobes" the data to the printer. As we mentioned previously, a "strobe" is a control signal that is used to load data into a hardware buffer. In this case, the hardware buffer is in the printer.

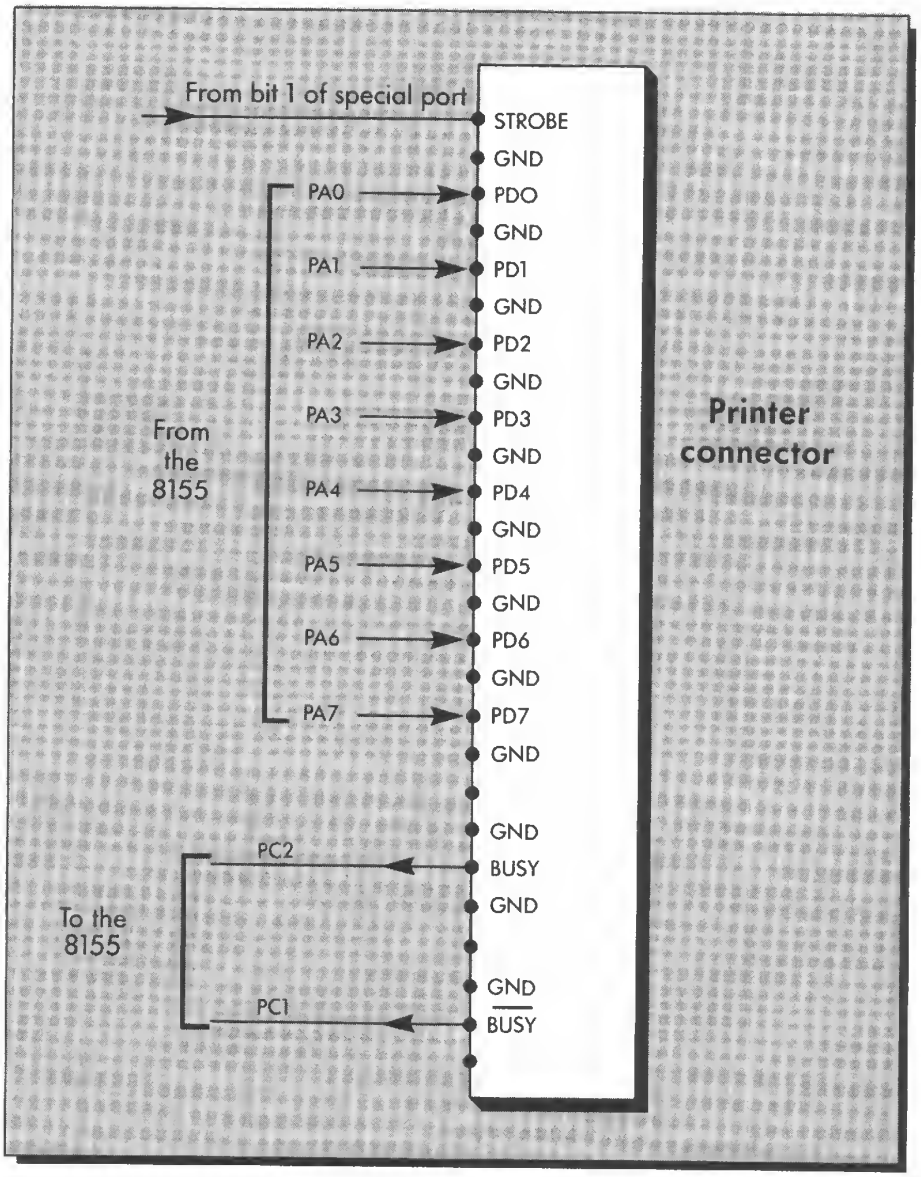


Figure 2-12. Printer interface

Summary

In this chapter we have explored the major hardware features of the Model 100. We have seen that it uses a CMOS version of the 8085 CPU, a member of a very popular family of chips. We have seen how the ROM, RAM, and ports are constructed and controlled, we have explored the multifunction 8155 chip, and we have quickly surveyed various subsystems such as the LCD, keyboard, and real time clock. Many of these subsystems have chapters devoted to them later in the book.

3

Hidden Powers of the ROM

Concepts
- Overall organization of ROM
- Interrupt entry points
- BASIC
- TELCOM
- MENU
- ADDRSS
- SCHEDL
- TEXT
- Cold and warm starts

In this chapter we will explore the secrets of the Model 100's ROM. We will study the overall organization of the ROM and then look at specific routines and tables that help run the BASIC, TELCOM, ADDRSS, SCHDL, TEXT, and MENU programs. The routines we study here will be those that perform general management tasks rather than controlling specific devices. The remaining chapters of this book will cover specific devices and their associated control routines.

There are 32 kilobytes of ROM in the Model 100. In these bytes are hundreds of useful routines that make the Model 100 a very powerful computer. We will start from the beginning (address 0) and work our way to the top of this ROM (address 7FFFh = 32767d) (see Figure 3-1).

At the lowest addresses of memory we find interrupt entry points. These are a series of hardware and software entry points to perform fundamental I/O and syntactical tasks.

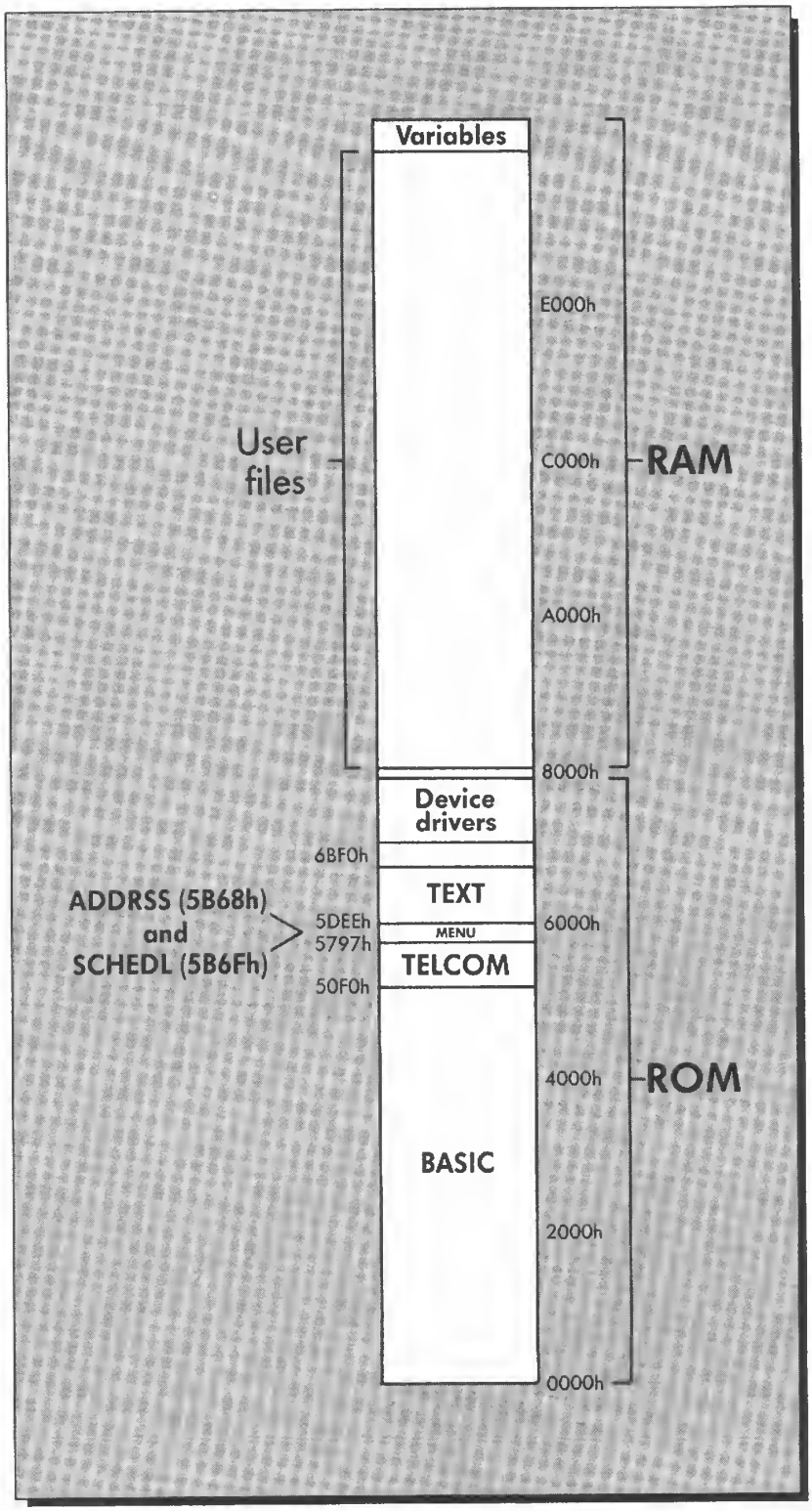


Figure 3-1. Layout of the Model 100's ROM

Next comes the BASIC interpreter. BASIC extends through a large area of ROM and includes areas for address and symbol tables, error handling, command input, command interpretation, and execution of individual commands.

The TELCOM program follows. It has routines that make the Model 100 into a terminal, including the ability to upload and download files. Next comes the MENU program, which has routines that allow you to move the cursor around the menu and dispatch to whatever program you select. The ADDRSS and SCHEDL programs are next. These programs blend into the TEXT program, which contains routines to manipulate the cursor and perform various editing functions. The final areas (highest addresses) of ROM contain initialization routines and primitive device control routines.

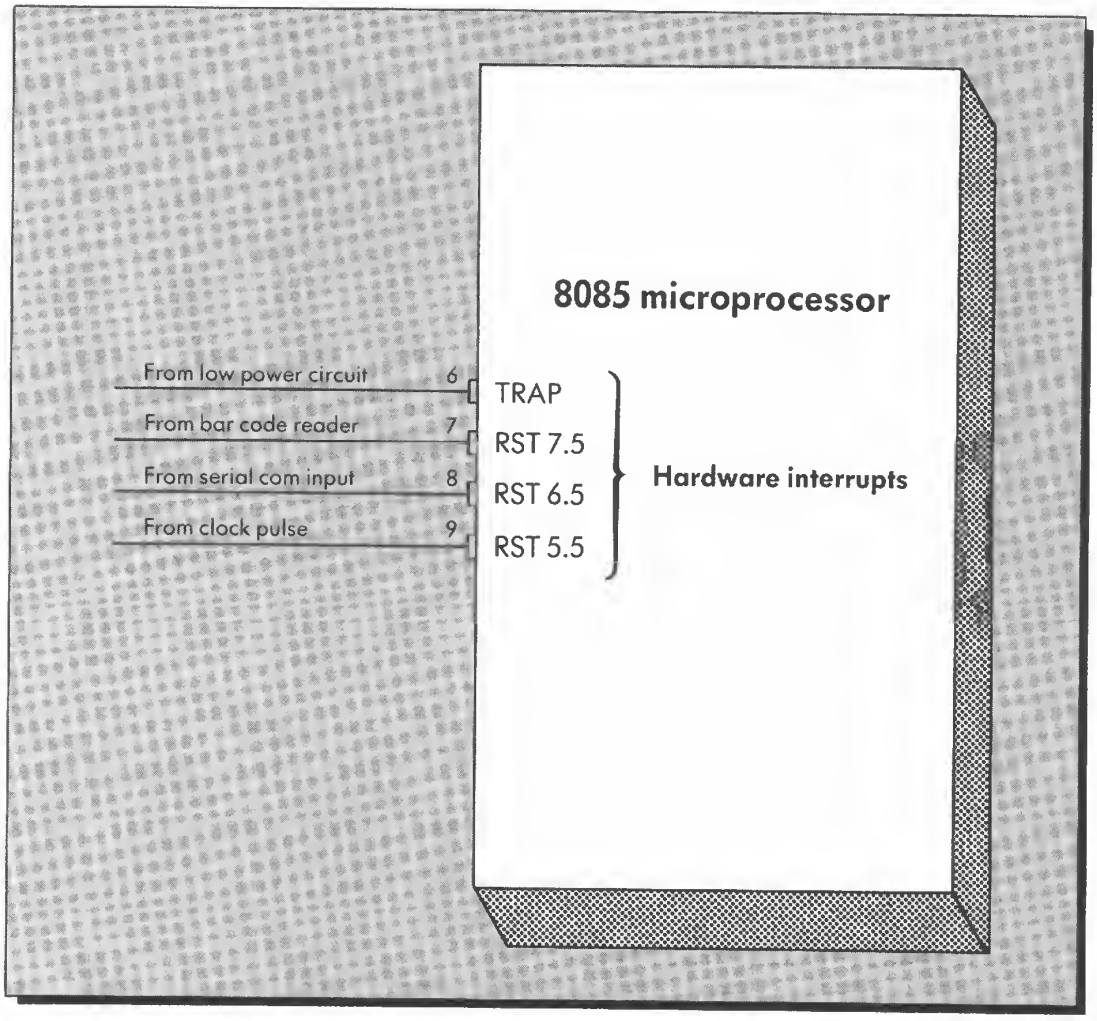


Figure 3-2. The hardware interrupt signal lines

We have strictly followed the physical organization of the ROM memory in this chapter in order to make the chapter easy to use as a reference guide to the ROM. However, this leads to a "bottom up" approach, particularly in studying BASIC. When you first read this chapter, you may want to start with the section on running BASIC programs and then go back and scan through the earlier sections as you need them.

The Interrupt Entry Points

The very lowest locations of ROM contain entry points for the 8085 CPU's software and hardware interrupt routines. The Model 100 uses all twelve of these special entry points. Eight of these are "called" by the one-byte RST (ReStarT) instructions. The other four are activated by special interrupt signal lines coming into the CPU from the low power detection circuit, the bar code reader interface, the serial communications line, and the real-time clock (see Figure 3-2).

Software Interrupts

Let's start with the software interrupts. These are eight locations with addresses that are even multiples of eight. Each one is called by one of the RST instructions. The Model 100 has placed special routines in these RST locations that in effect extend the instruction set of its 8085 CPU. For example, RST 3 compares the DE and HL register pairs (which would have been a handy CPU instruction if Intel had included it in the design of the 8085 CPU).

The RST 0 entry point is at location 0 and can be considered both a software and a hardware entry point. It is activated by the RST 0 instruction (software) and the RESET switch (hardware) on the back of the computer. The code at location 0 is a jump to location 7D33h = 32,051d, where there is a procedure to restart the Model 100 after it gets hung up.

Routine: RST 0

Purpose: To restart the Model 100 and return to main menu

Entry Point: 0h = 0d

Input: None

Output: The machine is reset.

BASIC Example:

```
CALL 0
```

Special Comments: None

The RST 1 entry point is located at 8h = 8d. Here there is a routine that is useful in analyzing the syntax of BASIC commands. This routine compares the next byte from the BASIC command line with the byte immediately following the RST 1. If the two bytes agree, it returns, continuing execution right after the byte following the RST 1 instruction. If the two disagree, it jumps to a routine that declares a syntax error and halts execution of the BASIC program.

Routine: RST 1

Purpose: To compare next byte of machine-language program with next byte of BASIC command line

Entry Point: 8h = 8d

Input: Upon entry, the HL register pair points to a byte in memory.

Output: If the byte in memory matches the next byte after the RST 1 instruction, then the routine returns, continuing execution right after the byte following the RST 1 instruction. If the two bytes disagree, a syntax error is declared and BASIC returns to its input mode.

BASIC Example: Not directly applicable

Special Comments: None

The RST 2 entry point is at location 10h = 16d. This jumps to location 858h = 2,136d, where there is a routine that is also used in analyzing the syntax of BASIC commands. This routine examines the next byte of the command line. If that byte contains the ASCII code of a digit from 0 to 9, the routine returns with the carry flag set. If not, it skips over spaces, tabs, and linefeeds in the command line. It returns once it finds a byte that is not one of these. If it finds a byte that is zero, it sets the Z flag. This usually means "end of command line".

Routine: RST 2

Purpose: To search for next byte of command, checking for zero byte or decimal digit

Entry Point: 10h = 16d

Input: Upon entry, the HL register pair points to a byte position in a BASIC command line.

Output: The routine skips over spaces and tabs in the command line. It stops when it finds a byte that is not a space or a tab. It returns with the value of this byte in the A register. If this byte contains the ASCII code of a decimal digit (0 through 9), the routine returns with the carry flag set. If this byte is zero, it returns with the Z flag set.

BASIC Example: Not directly applicable

Special Comments: None

The RST 3 entry point is at location 18h = 24d. Here there is a routine that compares the contents of the DE with the contents of the HL register. If they are equal, it returns with the Z flag set. If HL is less than DE, then the carry is set. Otherwise it is clear.

Routine: RST 3

Purpose: To compare two 16-bit integers

Entry Point: 18h = 24d

Input: Upon entry, the register pairs HL and DE each contain 16-bit integers.

Output: When the routine returns, the carry and zero flags are set as follows:

Condition	Flags
HL < DE	C and NZ
HL = DE	NC and Z
HL > DE	NC and NZ

BASIC Example: Not applicable

Special Comments: None

The RST 4 entry point is at location 20h = 32d. This jumps to location 4B44h = 19,268d, where there is a routine to print a character on the LCD screen (see Chapter 4).

Routine: RST 4

Purpose: To print a character on the LCD screen

Entry Point: 20h = 32d

Input: Upon entry, the A register contains the ASCII code of the character to be printed.

Output: The character is printed on the screen.

BASIC Example:

```
CALL 32,X
```

where the value of X is the ASCII code of the character.

Special Comments: None

The RST 5 entry point is at location 28h = 40d. This jumps to location 1069h = 4201d, where there is a routine that checks the data type of the expression currently being processed by BASIC.

Routine: RST 5

Purpose: To check the data type of the expression currently being processed

Entry Point: 28h = 40d

Input: Upon entry, the data type of the expression must be in location FB65h = 64,357d.

Output: Upon exit, the flags indicate the data type according to the following rules:

Data Type	Flag Condition
Integer	Zero flag clear (NZ)
String	Sign flag clear (P)
Single-precision real	Carry flag set (C)
Double-precision real	Parity flag set (E)

BASIC Example: Not directly applicable

Special Comments: None

The RST 6 entry point is at location 30h = 48d. This jumps to location 33DCh = 13,276d, where there is a routine that checks the sign of the number currently being processed by BASIC.

Routine: RST 6

Purpose: To get the sign of the real expression currently being evaluated

Entry Point: 30h = 48d

Input: Upon entry, a valid single- or double-precision real number must be in BASIC's accumulator (locations FC18h = 64,536d through FC1Fh = 64,543d).

Output: The A register is used to return a result. If the real number in BASIC's accumulator is zero, the result in A is zero. If the real number is negative, this result is −1. If the real number is positive, the result is +1.

BASIC Example: Not directly applicable

Special Comments: None

The RST 7 entry point is at location 38h = 56d. This jumps to location 7FD6h = 32,726d, where there is a dispatch routine that uses a special table of addresses stored in RAM (see Figure 3-3). This table is called the "hook" table because it gives programmers a chance to insert their own routines in strategic places in the ROM programs. In Chapter 4 we will see an example of the way the RST 7 instruction is used in the ROM programs.

The RST 7 routine uses the byte following the RST 7 instruction to look up an address in a table starting at location FADAh = 64,218d in RAM. The routine calls this subroutine and returns to continue execution right after the byte following the RST 7 instruction. The first 57 addresses point to a subroutine consisting of just a RETurn instruction, and the last 37 locations each point to a routine that declares an illegal function error.

Routine: RST 7

Purpose: To dispatch to routine in hook table

Entry Point: 38h = 56d

Input: The byte following the RST 7 instruction is used as input and must be between 0 and 56.

Output: The routine dispatches to the location whose address is at location FADAh = 64,218d plus twice the value of the byte following the RST 7 instruction.

BASIC Example: Not directly applicable

Special Comments: None

Hardware Interrupts

The four hardware interrupt entry points are located at 24h = 36d, 2Ch = 44d, 34h = 52d, and 3Ch = 60d. In each case there is a signal line coming into the CPU that can be used to cause the CPU to interrupt its normal activity and immediately branch to one of these entry points. The routines pointed to by these entry points are the key to understanding how the corresponding devices work.

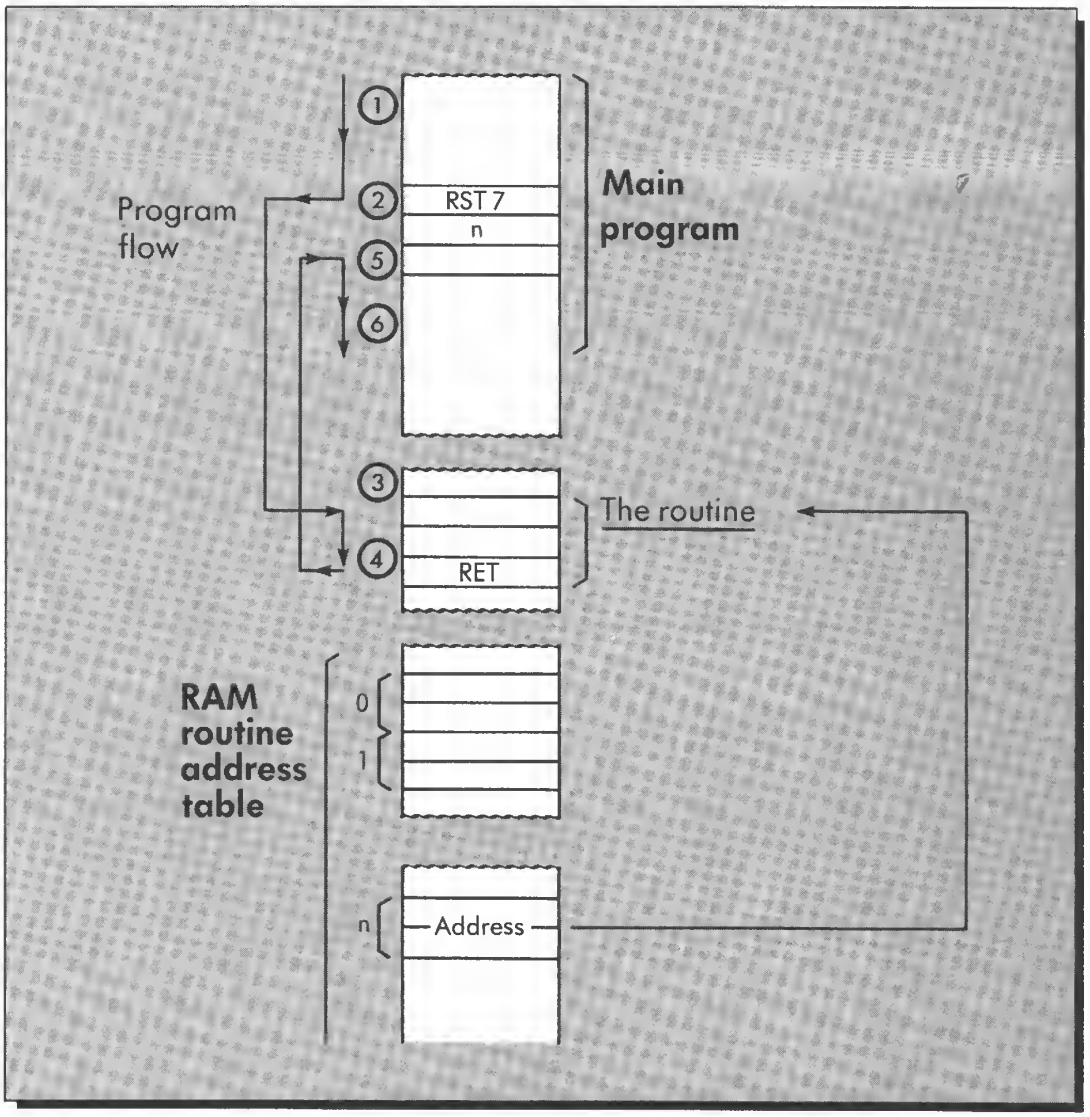


Figure 3-3. RAM routine table

The TRAP entry point is at location 24h = 36d. It is activated when the power circuit indicates a low-power condition. This entry point jumps to location F602h, where the low-power condition is handled.

Routine: TRAP

Purpose: To handle low-power condition

Entry Point: 24h = 36d

Input: None

Output: Shuts off the power.

BASIC Example:

```
CALL 36
```

Special Comments: To restore the power, turn the power switch (on right side of machine) off and then on again.

The RST 5.5 entry point is at location 2Ch = 44d. It is activated by the bar code reader. Here, interrupts are disabled with the DI instruction. Then there is a jump to location F5F9h = 62,969d, where the bar code reader interrupt service routine resides.

Routine: RST 5.5

Purpose: To receive data from bar code reader

Entry Point: 2Ch = 44d

Input: From the bar code reader

Output: To the Model 100

BASIC Example: Not directly applicable

Special Comments: Not implemented in the machine as it comes from the factory.

The RST 6.5 entry point is at location 34h = 52d. It is activated by input from the serial communications line. Here, interrupts are disabled with the DI instruction. Then there is a jump to 6DACh = 28,076d, where the interrupt service routine for accepting input from the serial communications line resides.

Routine: RST 6.5

Purpose: To input byte from serial communications line

Entry Point: 34h = 52d

Input: A byte from the serial communications line must be ready.

Output: To the serial communications input buffer

BASIC Example: Not applicable

Special Comments: None

The RST 7.5 entry point is at location 3Ch = 60d. It is activated by a pulse from the clock every 4 milliseconds. At this entry point, interrupts are disabled with the DI instruction. Then there is a jump to 1B32h = 6,962d, where the background task begins. We will discuss the background task in more detail in the next few chapters.

Routine: RST 7.5

Purpose: To initiate one cycle of the background task

Entry Point: 3Ch = 60d

Input: None

Output: Affects LCD display, real-time clock, and keyboard (see Chapters 4, 5, and 6).

BASIC Example: Not applicable

Special Comments: See chapters 4, 5, and 6 for a full discussion of the background task.

The BASIC Interpreter

The BASIC interpreter provides a friendly programming environment for the Model 100. BASIC is a popular language because it is powerful, has simple and direct syntax rules, and can immediately run programs or direct commands as they are entered or modified. In this section you will see exactly how all this works. We will also show you a simple program that makes BASIC even more powerful and friendly.

BASIC extends from about 40h = 64d to about 50F0h = 20,720d in the Model 100's ROM. However, many of the routines in this area are shared by other programs and BASIC uses some routines located in other areas.

BASIC Symbol and Address Tables

The address and symbol tables used by BASIC start at location 40h = 64d, right after the interrupt entry points. These tables are described briefly below. Refer to the appendices for their complete contents.

The first table (locations 40h = 64d through 7Fh = 127d) is a list of addresses of routines to handle various BASIC functions such as INT, ABS, SIN, and PEEK (see Appendix A). This table is used by the BASIC interpreter to help it find these routines.

The second table (locations 80h = 128d through 25Fh = 607d) contains all the BASIC keywords (see Appendix B). This table is used by the BASIC interpreter to convert its keywords into special one-byte codes called tokens. The tokens range in numerical value from 128 to 255. The first keyword in this table is assigned to the token value 128, the second is assigned to 129, and so on through the rest of the table.

The third table (locations 262h = 610d through 2E1h = 737d) contains the addresses of the routines to handle BASIC commands. These are given by the BASIC *initial* keywords — that is, keywords that appear at the beginning of a BASIC command line, such as FOR, LET, GOTO, and CLEAR (see Appendix C).

The fourth table (locations 2E2h = 738d through 2EDh = 749d) contains operator priorities for the binary operations: +, −, *, and so on (see Appendix D). These priorities allow BASIC to recognize which operations to do first in a complex algebraic expression. For example, in the expression A + B * C, the * operation should be performed first and the + second.

The fifth table (locations 2EEh = 750d through 2F7h = 759d) contains addresses of some numerical conversion routines (see Appendix E).

These convert numbers to double-precision, integer, and single-precision formats.

The sixth table (locations 2F8h = 760d through 303h = 771d) contains the addresses of the routines for binary operations of +, −, *, /, and comparison for double-precision floating-point numbers (see Appendix F).

The seventh table (locations 304h = 772d through 30Fh = 783d) contains the addresses of the routines for binary operations of +, −, *, /, and comparison for single-precision floating-point numbers (see Appendix G).

The eighth table (locations 310h = 784d through 31Bh = 795d) contains the addresses of the routines for binary operations of +, −, *, /, and comparison for integers (see Appendix H).

The ninth table (locations 31Ch = 796d through 359h = 857d) contains a list of all the two-character error designators (see Appendix I).

Area Mapped to High Memory

The contents of the next area of ROM (locations 35Ah = 858d through 3E9h = 1001d) are moved to RAM (locations F5F0h = 62,960d through F67Fh = 63,103d) when the Model 100 is first initialized. This area contains various variables and routines, including those that help control the keyboard, LCD screen, and BASIC itself. Their initial values are stored in ROM. When this area is mapped to RAM, these initial values are put in place.

Among the routines in this area are ones that are called at the beginning of certain interrupt service routines. When this area is moved to RAM, these routines consist of just a RETurn instruction followed by two NO oPeration instructions. However, because they are placed in RAM, they can be changed to JuMPs to your own interrupt routines.

Other routines in this area implement the INP and OUT commands (see boxes) and key steps of the line-drawing algorithm used in the LINE command (see Chapter 4).

Routine: INP — BASIC Command

Purpose: To input from port

Entry Point: F66Ah = 63,082d

Input: The address of the desired port must be in location F66Bh = 63,083d.

Output: The routine returns with the data from the port in the A register.

BASIC Example: Not directly applicable

Special Comments: When you use the INP function, BASIC automatically puts the port address in location F66Bh = 63,083d.

Routine: OUT — BASIC Command

Purpose: To output to a port

Entry Point: F667h = 63,079d

Input: Upon entry, the A register contains the data to be output, and location F668h = 63,080d contains the port address.

Output: The contents of the A register are sent out the port.

BASIC Example:

```
POKE 978,PORT
CALL 977,DATA
```

where PORT is the port address and DATA is the value of the data byte.

Special Comments: None

Messages and Errors

The next area of ROM (locations 3EAh = 1002d through 400h = 1024d) contains messages used by BASIC. These include "Error", "?", "Ok", and "Break".

A set of routines that handles various errors runs from about 422h = 1058d to 501h = 1281d.

The main entry point for handling errors is at location 45Dh = 1117d (see box). Upon entry, the E register must contain an error code. When this routine is called, it displays the corresponding error message and aborts execution of a BASIC program. The error codes are given on page 217 of the Model 100 owner's manual and in Appendix I.

Routine: Main BASIC Error Routine

Purpose: To display an error message

Entry Point: 45Dh = 1117d

Input: Upon input, the E register contains the error code.

Output: The routine displays the error message corresponding to the error code in the E register. See the error code table on page 217 of the owner's manual and Appendix I of this book.

BASIC Example: Not directly applicable

Special Comments: None

There are special error entry points in which the E register is loaded with a specific error code and then the main entry point processes the error. These special entry points are located in two areas of the ROM. The first area goes from 446h = 1094d to 45Ch = 1116d, and the second area goes from 504Eh = 20,558d to 506Ah = 20,586d (see Appendix J).

Command Entry

The main command entry loop for BASIC begins at location 501h = 1281d. We will discuss some of its highlights.

The code at location 50Bh = 1291d displays the message "Ok" on the screen. The message is stored at location 3F6h = 1014d, as discussed previously. The routine that displays a message is located at 27B1h = 10,161d. It expects the address of the message in the HL register pair.

At location 511h = 1297d, the value FFFFh (minus one in 2's complement arithmetic) is placed in the current BASIC line number F67Ah = 63,098d. This tells BASIC that it is in command entry or immediate mode rather than running a program.

Next the INLIN routine at 4644h = 17,988d is called (see box). This routine waits for the next line from the keyboard. If the line is nonempty, the first level of interpretation, called tokenizing, takes place.

Routine: INLIN

Purpose: To input a line from the keyboard

Entry Point: 4644h = 17,988d

Input: From the keyboard

Output: When the routine returns, the line is stored in memory starting at location F685h = 63,109d. The line is terminated in memory with a byte whose value is zero.

BASIC Example:

```
CALL 17988
```

Special Comments: You can pick up the line by using the PEEK statement.

During tokenizing, all the keywords in the BASIC command line are converted to their one-byte tokens (see BASIC keyword table — Appendix B). The command line is then in a very compact form, making it easier to run and to store.

The tokenizing routine is located at 646h = 1606d and is called from 54Ah = 1354d. The tokenizing routine has to be careful not to tokenize REMarks and quoted material, and it has to translate lowercase characters in variable names to uppercase.

Routine: Tokenize

Purpose: To tokenize a BASIC command line

Entry Point: 646h = 1606d

Input: Upon entry, the HL register pair points to (contains the address of) the untokenized line.

Output: When the routine returns, the tokenized line is stored in memory starting at location F681h = 63,105d. The tokenized line is terminated by a byte whose value is zero.

BASIC Example:

```
A0=VARPTR(A$)
A1=PEEK(A0+1)+256*PEEK(A0+2)
CALL 1606,0,A1
```

where A$ is a string containing the line to be tokenized, A0 is the address of the string's length and location parameters, and A1 is the address of the actual bytes of the string.

Special Comments: You can use PEEK to get the bytes of the tokenized string from where they are stored in memory.

The routine works by matching character strings in the command line against character strings in the token table at 80h = 128d (see Appendix B). When it is looking for a match, it sweeps through the table, counting all the mismatches until it finds a match. This count then is used to compute the value of the token.

After tokenizing the line, the BASIC interpreter tries to insert the newly created line into the program. It must first check whether the new line is an immediate command. This is done with the RST 2 instruction, which is invoked at 523h = 1315d, and the conditional jump at 552h = 1362d. If the line contains an immediate command, the command is executed immediately by jumping to location 4F1Ch = 20,252d.

If the line is to be inserted into the program, then the code from 555h = 1365d to 5ECh = 1516d is used to place it in the program. If a line with the same line number already exists in the program, the new line replaces the old; otherwise, it is simply inserted in the program. The routine to search for the line number in the program is located at 628h = 1576d (see box).

Routine: Search for Line Number

Purpose: To search a BASIC program for a line with a specified line number

Entry Point: 628h = 1576d

Input: Upon entry, the DE register pair has the line number in binary.

Output: When the routine returns, the HL register pair contains the address of the proper location to insert a line with the specified line number into the program. If the specified line number matches an existing line number in the program, the carry flag is clear; otherwise it is set.

BASIC Example: Not directly applicable

Special Comments: None

Running BASIC Programs

The next major area of memory contains the code that runs BASIC programs. This is the heart of the BASIC interpreter. This section extends from about 804h = 2052d to 871h = 2161d.

This code has to check several conditions as it keeps running your program. The first thing it checks is the communications line, by calling a routine located at 6D6Dh = 28,013d (see Chapter 7). Then it checks for an interrupt from the real-time clock by calling a routine at 4028h = 16,424d (see Chapter 5). Next it checks for a break from the keyboard by calling a routine at 13F3h = 5107d (see Chapter 6).

The BASIC interpreter then looks for a colon indicating the next statement of a multiple statement on a BASIC command line. Finally, it checks for the end of the program. It returns to the command loop via a jump to 428h = 1064d if the program has indeed ended.

If the program has not ended, interpretation continues. The current line number is stored in location F67Ah = 63,098d, and then the interpreter dispatches to the routine to handle the keyword for the BASIC command currently under consideration. It points to the address table for BASIC commands. It is interesting to note that the last part of the dispatching routine (locations 858h = 2136d through 871h = 2161d) uses the same code as is used by the RST 2 routine to check for numbers and skip over spaces.

BASIC Keyword Routines

The next area of ROM contains routines to handle the various BASIC keywords. This area extends from about 872h = 2162d to about 50F0h = 20,720d. It also includes some isolated sections of code before and after this. For example, the code for the FOR statement extends from 726h = 1830d to about 803h = 2051d, which is between the tokenizer routine and the command dispatcher.

The LET Statement

Among the most fundamental parts of the code for the Model 100 is the code for handling the LET command (see box). Each LET command statement consists of a variable name or identifier followed by an equal sign, followed by a BASIC expression. Executing the LET command involves locating variables and interpreting BASIC expressions. In a sense it is the key to understanding how BASIC works in the Model 100.

Routine: LET — BASIC Command

Purpose: To evaluate BASIC expressions and store the results in the indicated BASIC variables

Entry Point: 9C3h = 2499d

Input: Upon entry, the HL register pair points to (contains the address of) the rest of the BASIC LET command line (starting with the variable identifier on the left of the "=").

Output: When the routine returns, the value of the BASIC expression is stored in the indicated variable (on the left side of the "=" in the LET command line).

BASIC Example:

```
CALL 2499,0,63105
```

where the input buffer at F681h = 63,105d contains a tokenized BASIC LET command line starting with the name of the variable on the left side of the "=". Call the tokenizer routine at 646h = 1606d before using this example.

Special Comments: The input buffer at 63,105 is also used by the INPUT statement.

The code for the LET routine extends from 9C3h = 2499d to A2Eh = 2606d.

We have included a BASIC program that calls this LET routine. It also calls the tokenizer routine at 646h = 1606d. This program turns your Model 100 into a fancy calculator. It can also be modified to turn the Model 100 into an interactive function grapher.

When you run this program, you are asked for the formula of the function. You should enter it as a BASIC formula F(X) in the single variable X. Next you are asked to give a starting value, an ending value, and a step size for X. Once you have entered these, the program displays the values of X and F(X) for the range and step size that you selected.

```
100 ' INTERACTIVE FUNCTION EVALUATOR
110 '
120   CLEAR 100
130 '
140   PRINT "INPUT A BASIC FORMULA";
150   PRINT " IN X"
160   INPUT "F(X)=";B$
170   A$= "Y="+B$+CHR$(0)
180 '
190   INPUT "STARTING VALUE OF X";X0
200   INPUT "ENDING VALUE OF   X";X1
210   INPUT "STEP SIZE OF X     ";X2
220 '
230 ' TOKENIZE THE FORMULA
240   A0 = VARPTR(A$)
250   A1 = PEEK(A0+1)+256*PEEK(A0+2)
260   CALL 1606,0,A1
270 '
280 ' DISPLAY LOOP
290   FOR X = X0 TO X1 STEP X2
300    CALL 2499,0,63105
310    PRINT X,Y
320   NEXT X
```

Let's look at the program more closely. On lines 140-170, the formula is input into the string B$ and packed into the string A$. The string A$ includes a prefix of "Y=" to supply the variable to the left of the "=" as well as the "=". There is also a zero character packed into the end of A$ to terminate the string for the ROM routines that we will call.

On lines 190-210, the starting step, ending step, and step size are input as variables X0, X1, and X2. Lines 230-250 compute the address where the actual bytes of the command string are stored in memory. The tokenizer routine at 646h = 1606d is called in line 260. Here, HL points to the command string for our custom LET statement. The display loop for the values

of X and F(X) runs from lines 280-320. It consists of a FOR loop indexed by the variable X with starting step, ending step, and step size determined by X0, X1, X2. Inside the FOR loop, the LET routine at 9C3h = 2499d is called to evaluate the formula Y = F(X), and then the values of X and Y are printed on the LCD screen.

To modify this program into an interactive function grapher, replace the PRINT command in the FOR loop by a line plotting command. Of course, you will have to add more controls to keep the screen looking good for the display.

Now let's return to our general discussion of the LET command and see how it is implemented in ROM.

The LET statement is the only BASIC command that doesn't require a keyword, although it also accepts lines that begin with the token for the keyword LET. As part of the command interpreter, BASIC must be able to detect commands that don't begin with a token. This function occurs at 840h = 2112d, just before dispatch to the individual routines for keywords. Here 80h = 128d is subtracted from the initial character on the line. If the result is negative, the character is not a token, and a jump is made to the code for LET.

Let's now look at the code for LET in more detail. It first calls a routine at 4790h = 18,320d, which returns the address of the variable on the left side of the equals sign (see box). Then it checks for the equals sign, aborting the program and declaring a syntax error if the equals sign is not present. It next calls a routine at DABh = 3499d to evaluate the right-hand side of the equals sign. Finally, it moves the computed value into the location reserved for the variable on the left of the equals sign.

Routine: Address Finder for BASIC Variables

Purpose: To locate the address of a BASIC variable

Entry Point: 4790h = 18,320d

Input: Upon entry, the HL register pair points to (contains the address of) the name of a variable.

Output: When the routine returns, the DE register pair points to the location of the variable in BASIC's table of variables. Information such as its data type and value is stored here.

BASIC Example: Not directly applicable

Special Comments: None

The address finder routine at 4790h = 18,320d expects the address of the name of the variable in the HL register pair, and it returns the address of the variable in the DE register pair.

The address finder routine is used by the VARPTR function as well as the LET command. VARPTR is a BASIC function that returns the address of a variable whose name is specified or a file whose number is specified. The code for the VARPTR function starts at F7Eh = 3966d. The part of the code that deals with the address of a variable begins at F92h = 3986d. It calls a routine at 482Ch = 18,476d that directly calls the address finder routine.

Routine: VARPTR — BASIC Function

Purpose: To return the location of a variable or a file

Entry Point: F7Eh = 3966d

Input: Upon input, the HL register pair points to (contains the address of) the name of a BASIC variable or a BASIC file number.

Output: When the routine returns, the address of the variable or the file buffer is in the DE register pair.

BASIC Example:

```
A = VARPTR(X)
```

where X is a BASIC variable and A is the BASIC variable where the address of X will be stored.

Special Comments: Cannot be called from BASIC because the result is returned in DE.

The address finder routine at 4790h = 18,320d first checks the first character in the name of the variable. If this is not between A and Z, then it aborts, declaring a syntax error; otherwise it puts the first character of the name in the C register. Next it picks up the second character, if present, and puts it into the B register. If there is no second character in the name, zero is placed in the B register.

Next, the routine skips through the rest of the name. After reaching the end of the name, it tries to determine the *type* of the variable — that is, whether it is an integer, single-precision real, double-precision real, or string variable.

There are two ways that the type can be determined. One is by means of special symbols such as %, $, !, and #. The other is by declaration through the DEFINT, DEFSNG, or DEFDBL statements. This last method specifies declared default data types according to the first character of the name. If one of the special symbols (%, $, !, or #) is not found, the address finder routine looks in a table at FB79h = 64,377d for the declared type (see Figure 3-4). No matter which way the type is determined, it is stored in location FB65h = 64,357d.

The routine then checks for the telltale parentheses of a variable that is an array entry. If the parentheses are present, it has to compute the position of the entry within the array. We won't describe this process, but it begins at 488Dh = 18,573d.

Next, the address finder routine searches for the name of the variable among the existing variables. Only the first two characters of the name are used. As we mentioned above, the C register contains the first character,

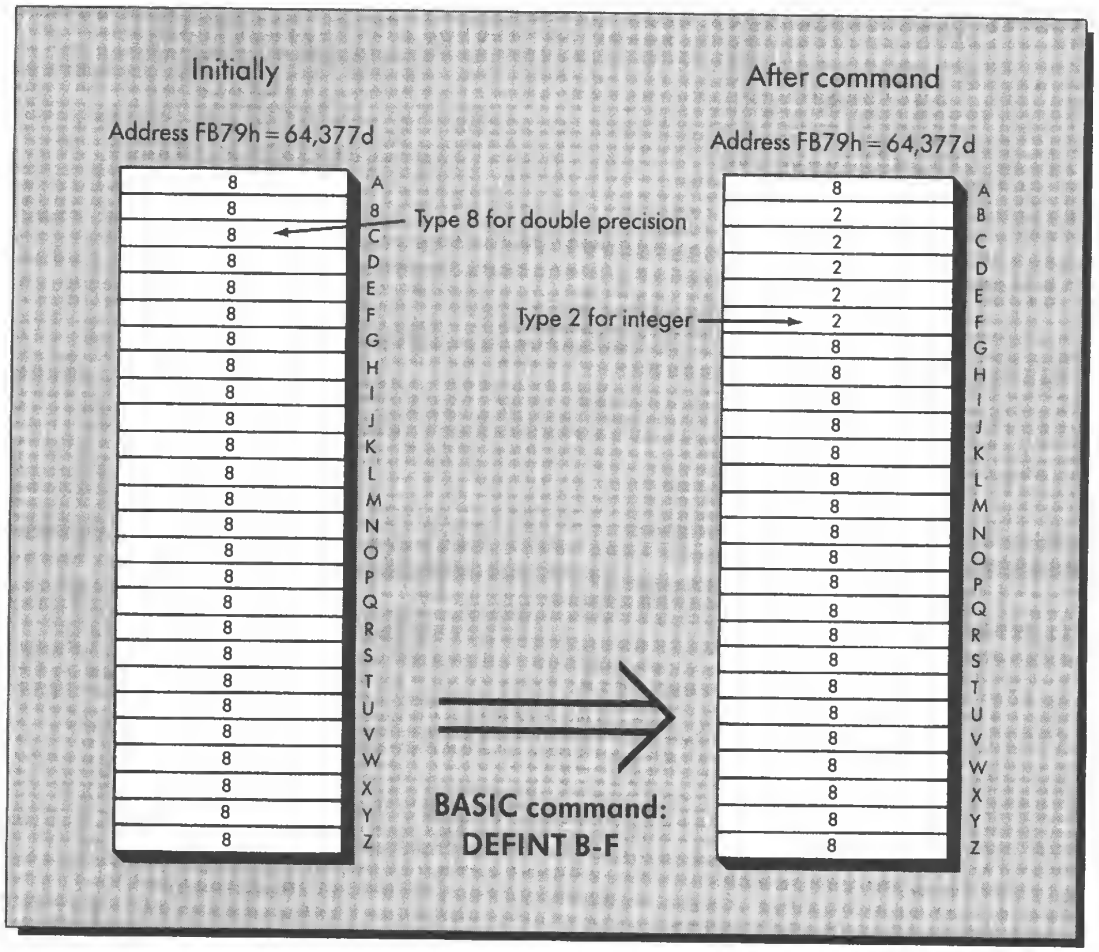


Figure 3-4. Declared default types

and the B register contains the second character, if it is present. The code for this search runs from 4801h = 18,433d to 4828h = 18,472d.

BASIC maintains its variables in a table whose address is stored at FBB2h = 64,434d (see Figure 3-5). If the address finder routine cannot find the variable in this table, it makes a place for the variable in the table. The code for doing this runs from 4835h = 18,485d to about 4875h = 18,549d.

The different types of variables require different amounts of storage in this table. Integer variables require 5 bytes, string variables require 6 bytes, single-precision real numbers require 7 bytes, and double-precision real numbers require 11 bytes. In each case, the first byte of its entry in the table contains a code for the type: 2 for integers, 3 for string variables, 4 for

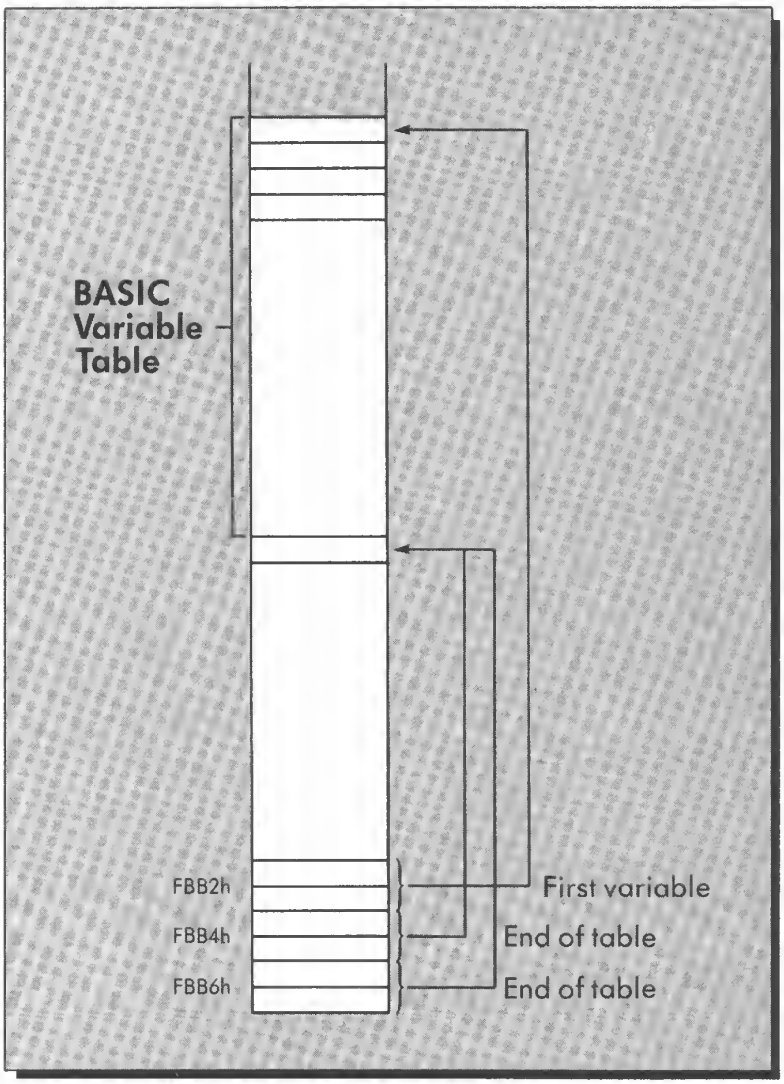


Figure 3-5. BASIC variable table

single-precision real numbers, and 8 for double-precision real numbers. You can see that the code for the type is 3 less than the number of bytes required in the table. For each of these types, the second and third bytes contain the ASCII code for the first character and second character in the name of the variable. If the name consists of only one character, a value of zero is used for the second character.

Now let's examine in detail how each type is stored (see Figure 3-6).

For an integer variable, the fourth and fifth bytes of its entry in the table contain the value of the integer in 16-bit two's complement binary form.

For string variables, the fourth byte contains the length of the string, and the fifth and sixth bytes contain the address of the location where the string is stored in memory.

For single- and double-precision real variables, the fourth through the last bytes contain the value of the number in a floating-point format. Floating point is like scientific format in that there is a sign, an exponent, and a

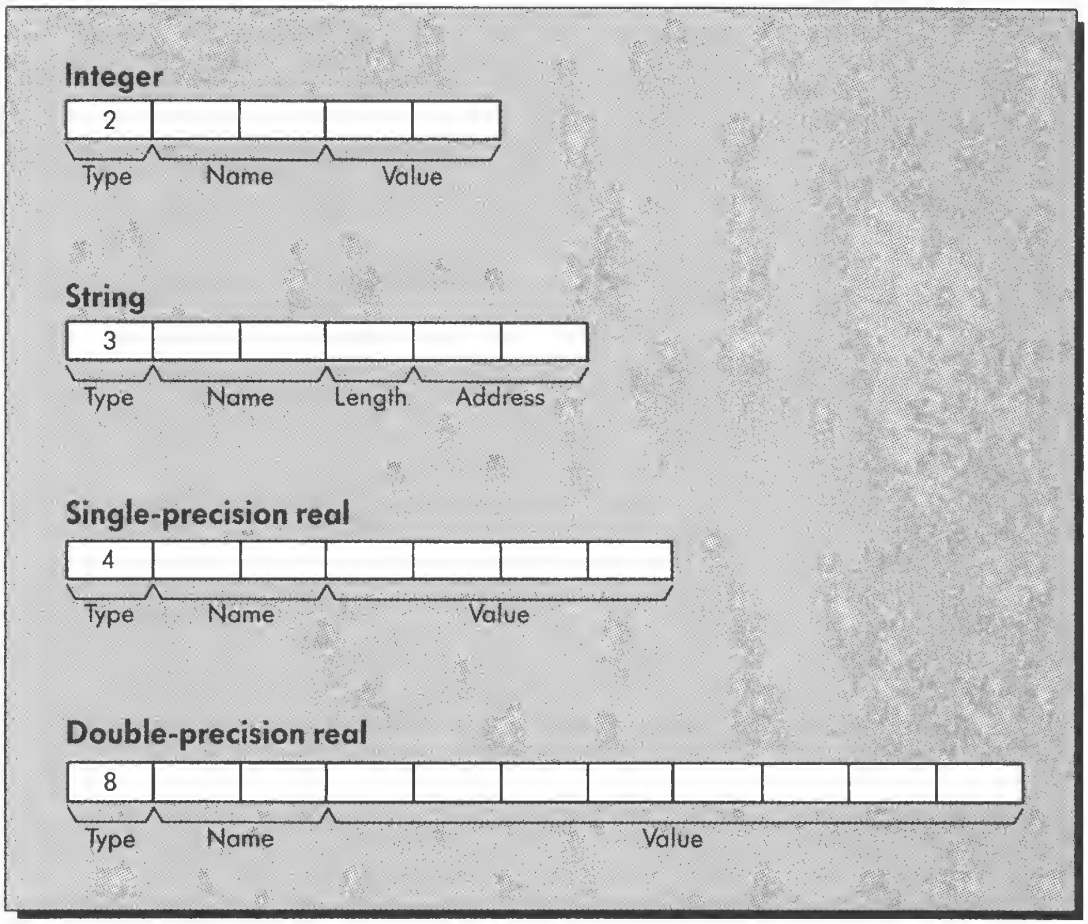


Figure 3-6. Storage allocation for BASIC variable

mantissa. We will describe how each of these is stored in the Model 100's floating-point format.

The sign of the number is stored in bit 7 of the first byte. A bit value of zero indicates a nonnegative number, and a bit value of 1 indicates a negative number.

The exponent of the number is stored in bits 0 through 6 of the first byte, with a bias of 64. This means that the actual exponent is obtained by subtracting 64 from the value stored in these bits.

The mantissa is stored in BCD form. That is, each decimal digit is represented by a nibble (four bits). For single precision there are six BCD digits in the three remaining bytes, and for double precision there are fourteen BCD digits in the remaining seven bytes. In either case, the decimal point is to the left of the most significant BCD digit.

Let's look at a couple of examples: the numbers 1.7 and -1.7. For number 1.7, the sign bit is 0, the actual exponent is 1, and the mantissa has the BCD digits 1 and 7 followed by zeros. This gives the expansion shown in Figure 3-7 for the four bytes allotted to the number.

For −1.7, the only difference is that the sign bit is 1. The other bits are the same.

Now let's look at the routine to evaluate the right side of the equals sign. This expression evaluator routine is located at DABh = 3499d (see box).

Routine: BASIC Expression Evaluator

Purpose: To evaluate BASIC expressions

Entry Point: DABh = 3499d

Input: Upon entry, the HL register pair points to (contains the address of) a tokenized BASIC expression.

Output: When the program returns, the value of the expression is contained in BASIC's accumulator (locations FC18h = 64,536d to FC1Fh = 64,543d).

BASIC Example: Not directly applicable

Special Comments: None

The expression evaluator routine attacks an expression by assuming that the expression can be written as two expressions connected by a binary operation such as +, −, *, /, ^, or comparison. It assumes that the value of the expression on the left has already been determined. It loops around and

around, trying to process a new binary expression each time. The first time through the loop, it calls a routine at F1Ch = 3868d to get the initial value on the left (see box).

Routine: BASIC Function Finder

Purpose: To evaluate unary expressions

Entry Point: F1Ch = 3868d

Input: Upon entry, the HL register pair points to (contains the address of) a term or function call of an expression in a tokenized BASIC command line.

Output: When the routine returns, the value of the term or function is in BASIC's accumulator (locations FC18h = 64,536d through FC1Fh = 64,543d).

BASIC Example: Not directly applicable

Special Comments: None

The expression evaluator uses the system stack to facilitate its operation. Each time through the loop it compares the priority of the current binary operation with the priority of the previous binary operation. If the old operation has higher priority, it is actually performed and then stored in an area of memory that acts as an accumulator; otherwise, all the information to be performed is pushed onto the stack. For some expressions, the priorities are such that many operations remain on the stack, but eventually all operations are performed.

A section of code running from DE6h = 3558d to E28h = 3624d pushes the needed information onto the system stack. This information includes the value and type of the expression on the left, the code for the operation, and a code for its priority. The priorities of these operations are contained

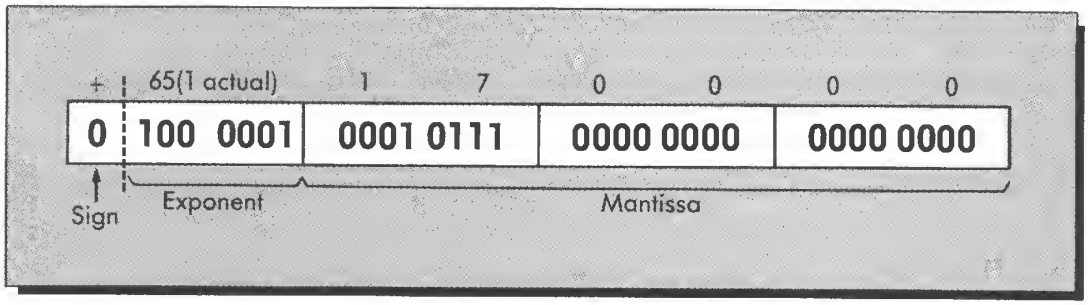


Figure 3-7. Floating point representation for 1.7

in a table located at 2E2h = 738d. The routine also pushes two addresses of its own code onto the stack.

The locations FC18h = 64,536d through FC1Fh = 64,543d are used as an accumulator for the expression evaluator. For integers, just FC1Ah = 64,538d and FC1Bh = 64,539d are used; for single precision, locations FC18h = 64,536d through FC1Bh = 64,539d are used; and for double precision, all eight locations are used (see Figure 3-8).

Let's look at a fairly simple example. This example will illustrate how the stack and the accumulator are used and how special starting and ending conditions are handled.

Suppose we wish to evaluate the expression:

2 + 3 * 5 + 4

You should look at Figure 3-9 during the following explanation. The leftmost expression is 2. Its value is put in the accumulator. The first binary operation is +. The first binary operation is always pushed. First the value in the accumulator, and then the operation, are pushed onto the stack. The next value, 3, is then evaluated in the accumulator. The next operation is *. It has a higher priority than +, so the value 3 from the accumulator and the * operation are also pushed onto the stack. Next, the value 5 is processed in the accumulator. The next operation is +, which has a lower priority than the previous operation (*), so evaluation of the * is begun. The computation 3 * 5 is performed and the result left in the accumulator. At this point the stack contains the 2 and the +, and 15 is in the accumulator. The next operation is +, which has the same priority as the + on the stack. Thus the operation on the stack is performed, yielding 17 in the accumulator. Since no more operations are pending, the value 17 and then the + operation are pushed onto the stack. Finally, the 4 is processed into the accumulator. Since the + is the last operation, it is processed, yielding a result of 21 in the accumulator.

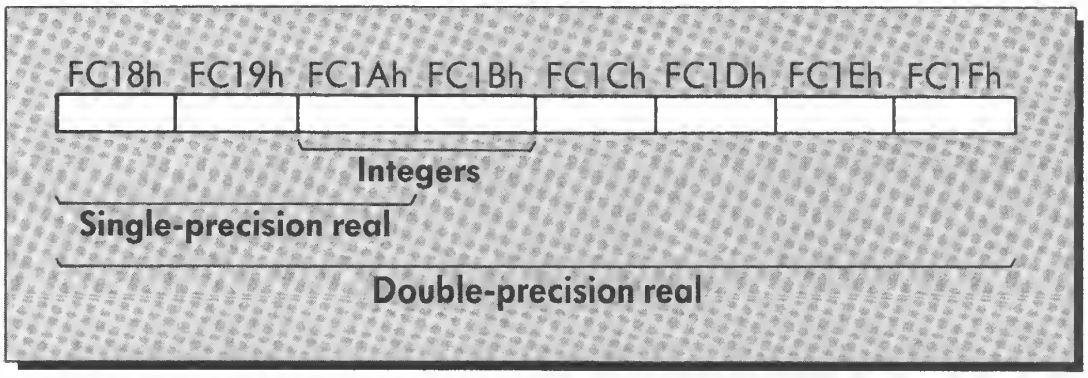


Figure 3-8. BASIC's accumulator

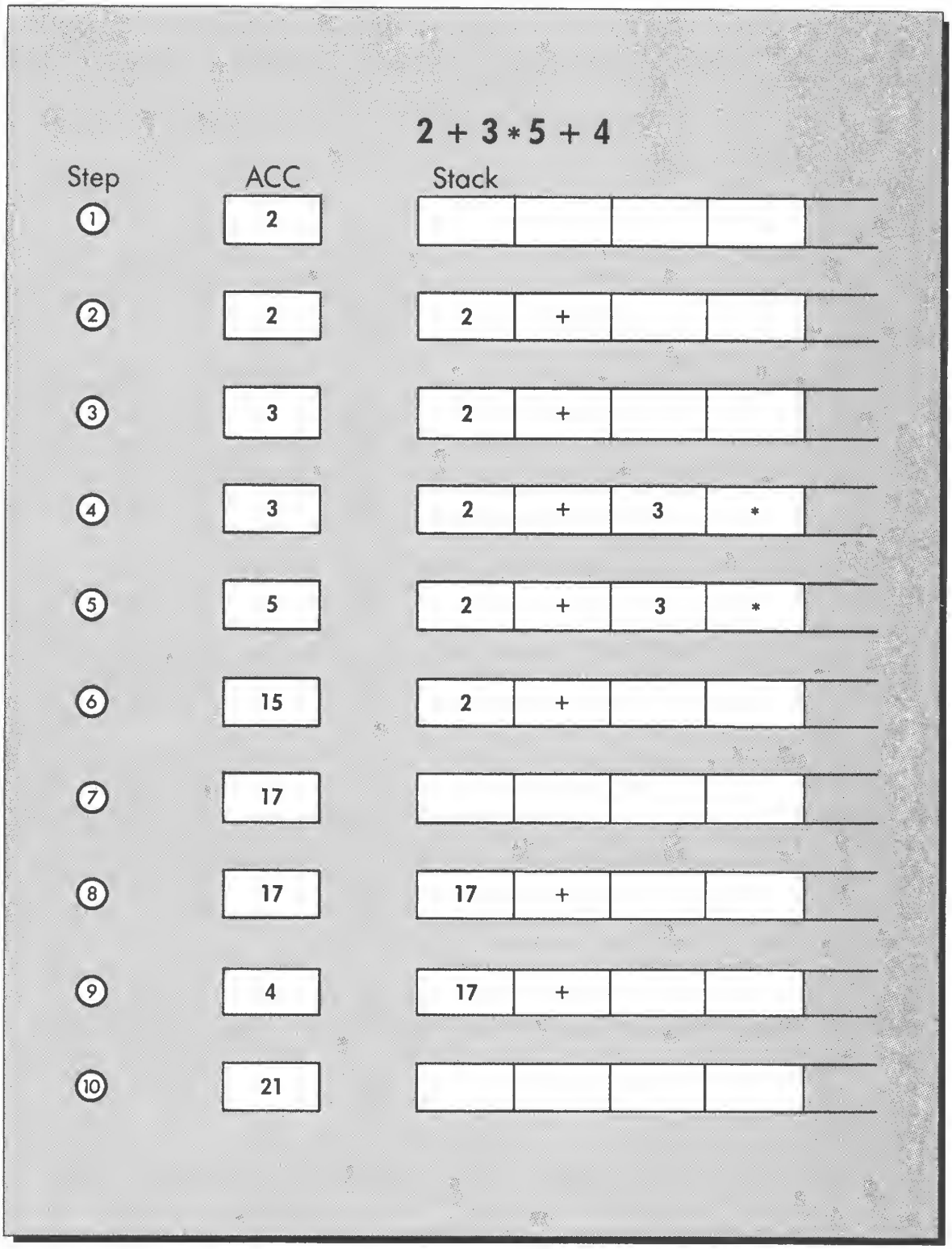


Figure 3-9. Evaluating an expression

The binary operations are performed starting at location E6Ch = 3692d. The code for the operation (+, −, *, and so on) is stored at location FB66h = 64,358d. The types are checked for the left and right sides. If they are not in agreement, various numerical conversion routines are called to make sure that the types do match. The tables in low ROM described earlier are used to dispatch to the appropriate operation.

Now let's look at the routine at F1Ch = 3868d, which evaluates single expressions. It uses the RST 2 call to skip spaces and look for numerical values. If it finds a numerical value, it jumps to 3840h = 14,400d, where the number is converted to the appropriate internal format. It next checks for a variable by calling the routine at 40F2h = 16,626d, which checks whether the next character is in the range from A to Z. If it is, the routine jumps to location FDAh = 4058d, where it calls the routine at 4790h = 18,320d to search for the variable among the existing variables. It gets the value of this variable and returns. The routine at F1Ch = 3868d continues, checking for such conditions as quotes, minus, NOT, and other unary operations and functions. It also looks for parentheses. It jumps to the appropriate code to handle whatever condition it finds.

Following this loop as it goes from F1Ch = 3868d through F46h = 3910d, to F51h = 3921d through F55h = 3925d, to F60h = 3936d through F7Dh = 3965d, to FA3h = 4003d through FCBh = 4043d, will allow you to see how the BASIC intepreter tries to detect all the possible "unary" operations.

The TELCOM Program

The TELCOM program allows you to use the Model 100 as a smart terminal for another computer either through an RS-232C communications line or through a telephone connection. You can even use it to upload and download text files.

The code for the TELCOM program runs from 5146h = 20,806d to about 5796h = 22,422d (see box). The main command input loop runs from 5152h = 20,818d to 5177h = 20,855d. The commands are input through the routine at location 4644h = 17,988d, which is called at location 516Ah = 20,842d. Dispatching is done via a routine at 6CA7h = 27,815d, which is called at 5175h = 20,853d. A table showing the names and addresses of the TELCOM commands starts at 5185h = 20,869d (see Table 3-1).

Routine: TELCOM

Purpose: To handle telecommunications

Entry Point: 5146h = 20,806d

Input: None

Output: Enters the TELCOM program

BASIC Example:

```
CALL 20806
```

Special Comments: The routine does not return to BASIC.

One of the TELCOM commands is TERM, which puts you into terminal mode (see box). Here is the BASIC command that will take you directly from BASIC to TERM:

```
CALL 21589
```

This is useful if you have just saved a BASIC file to another computer over the RS-232C communications line, using the SAVE "COM:... command in BASIC, and want to reestablish two-way communications with the other computer.

Command	Address
STAT	5100h = 20,736d
TERM	5455h = 21,589d
CALL	522Fh = 21,039d
FIND	524Dh = 21,069d
MENU	5797h = 22,423d

Table 3-1. TELCOM commands

Routine: TERM — a TELCOM mode

Purpose: To establish terminal mode for telecommunications

Entry Point: 5455h = 21,589d

Input: None

Output: Enters terminal mode of the TELCOM program.

BASIC Example:

```
CALL 21589
```

Special Comments: The routine does not return to BASIC.

The TERM mode has several commands. A "dispatcher" routine located at 54FCh = 21,756d (see Table 3-1) and a table starting at 550Dh = 21,773d are used to branch to the routines for each of these commands.

In Chapter 7 we will study the serial communications devices and the ROM routines that run them.

The MENU Program

The MENU program allows you to see what files you have in your Model 100 and select a particular one for editing or execution. It displays the main menu, which shows a directory of the files. The names of the files are displayed in six rows with four files to a row, giving a total of twenty-four possible files. You can select a particular file either by typing its name or by placing the cursor over the name in the directory and pressing ENTER.

The code for the MENU program extends from 5797h = 22,423d to about 5B67h = 23,399d. It consists of an initialization section, a main command loop, and a number of routines to handle the various cursor and dispatching commands. The command loop gets keystrokes from the user and interprets them as cursor and selection commands.

Routine: MENU

Purpose: To display the menu directory and select program or file

Entry Point: 5797h = 22,423d

Input: None

Output: The directory is displayed on the screen.

BASIC Example:

```
CALL 22423
```

Special Comments: This CALL will cause you to exit BASIC and return to the main menu.

The File Directory

The directory is displayed on the LCD screen as part of the initialization stage of the MENU program. There are a total of twenty-seven possible entries in the RAM directory. However, there are only twenty-four spots for entries in the LCD menu display. The other three entries are false entries; two secretly store the names of those who helped develop the Model 100.

The directory is stored in RAM starting at F962h = 63,842d (see Figure 3-10). Each entry in the RAM directory takes up eleven bytes. The first byte contains the file type and protection code (see Figure 3-11). The individual bits are assigned as follows: bit 7 is 1 if the entry is currently being used and 0 if it is invalid, bit 6 is 1 if it is an ASCII (DO) file, bit 5 is 1 if it is a machine-language file (CO), bit 4 is 1 if it is a ROM file, and bit 3 is 1 if it is an invisible file. Thus, for example, the ROM programs BASIC, TEXT, TELCOM, ADDRSS, and SCHEDL are stored as files of type B0h = 176d (valid, machine language, and ROM bits are all on). The second and third bytes contain the address of the body of the file in the usual low/high format. The next eight bytes contain the name of the file. For files created by the Model 100's various utilities, the first six bytes contain the main part of the name and the last two bytes contain the file extension. If the main part of the name is less than six characters, the remaining bytes between the main part of the name and the file extension are filled with spaces.

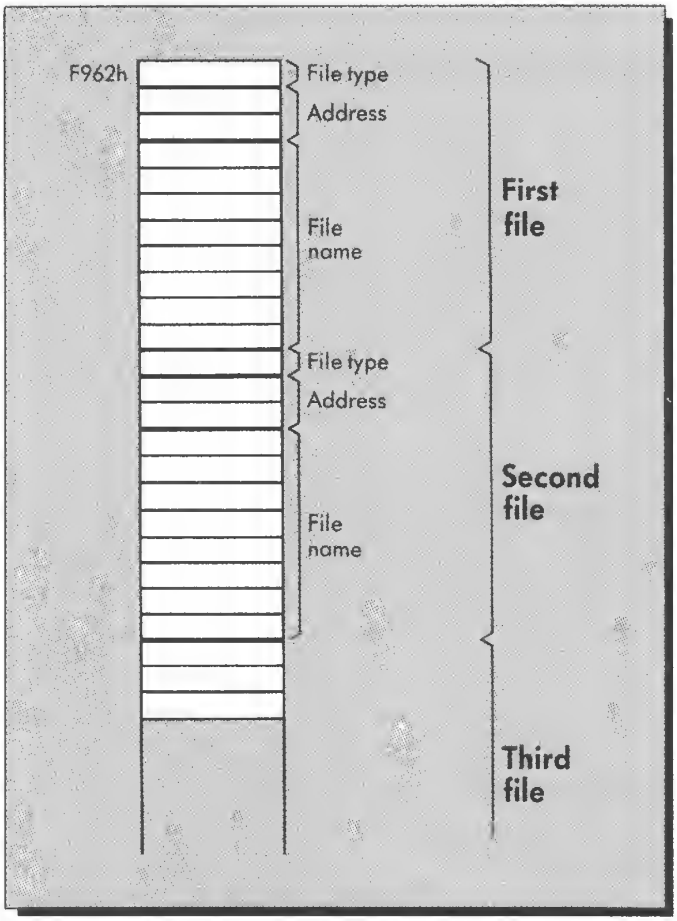


Figure 3-10. The directory

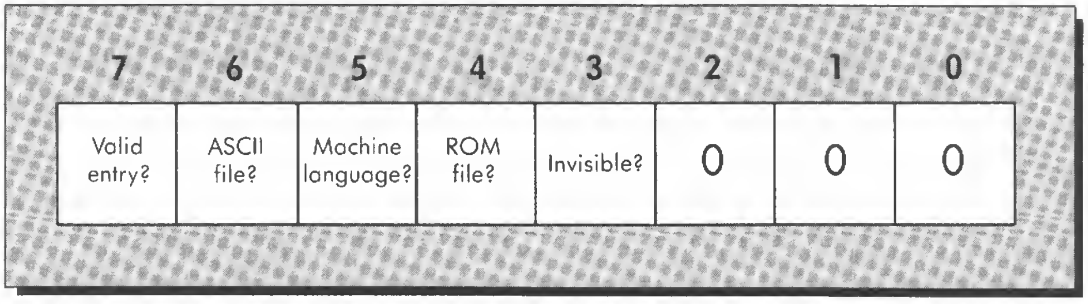


Figure 3-11. The bits of the file type byte

The following BASIC program displays the directory, giving file type and address for each file. It even shows the hidden files.

```
100 ' DISPLAY DIRECTORY
110 '
120   FOR I = 63842 TO 64138 STEP 11
130    PRINT USING "###"; PEEK(I);
140    ADR = PEEK(I+1)+256*PEEK(I+2)
150    PRINT USING "######";ADR;"   "
160    FOR K=3 TO 10
170     PRINT CHR$(PEEK(I+K));
180    NEXT K
190    PRINT
200   NEXT I
```

The program is straightforward. It consists of a loop that goes through all the directory entries, displaying the numbers and text as described above.

Now let's look at the MENU program in ROM. It is located at 5797h = 22,423d. It begins by setting up the screen for display of the menu. It turns off the reverse characters, cursor, and function key display, and it unlocks the display. It displays the time and date and the Microsoft copyright notice.

The main loop for displaying directory entries begins at 57F8h = 22,520d. Just before the loop, the DE register pair is loaded with the address of a short table located at 5B1Eh = 23,326d (See Figure 3-12). This table contains the following list of file types: B0h = 176d, F0h = 240d, C0h = 192d, 80h = 128d, and A0h = 160d. The table terminates with a zero. This routine will display directory entries with only these file types, and it will display them in this order. The first type (B0h = 175d) corresponds to the ROM program files such as BASIC, TEXT, and TELCOM. If you look at your display, you will see that they are always listed first. The third type (C0h = 192d) corresponds to regular ASCII files (file extension DO), and the fourth type (80h = 128d) corresponds to regular BASIC program files (file extension BA). You will notice that all ASCII (DO) files are listed before all BASIC (BA) files.

The main loop of the directory display gets a file type from the table, places it in the C register, and calls a routine at 5970h = 22,896d to display all directory entries of that file type. The loop continues until all the files of these five displayable types are displayed on the LCD screen.

The routine at 5970h = 22,896d loads the DE register pair with the beginning address of the directory (F962h = 63,842d) and the B register with the value 27, which is the total number of directory entries. It then goes into a loop that cycles through the entries, picking up the file type and

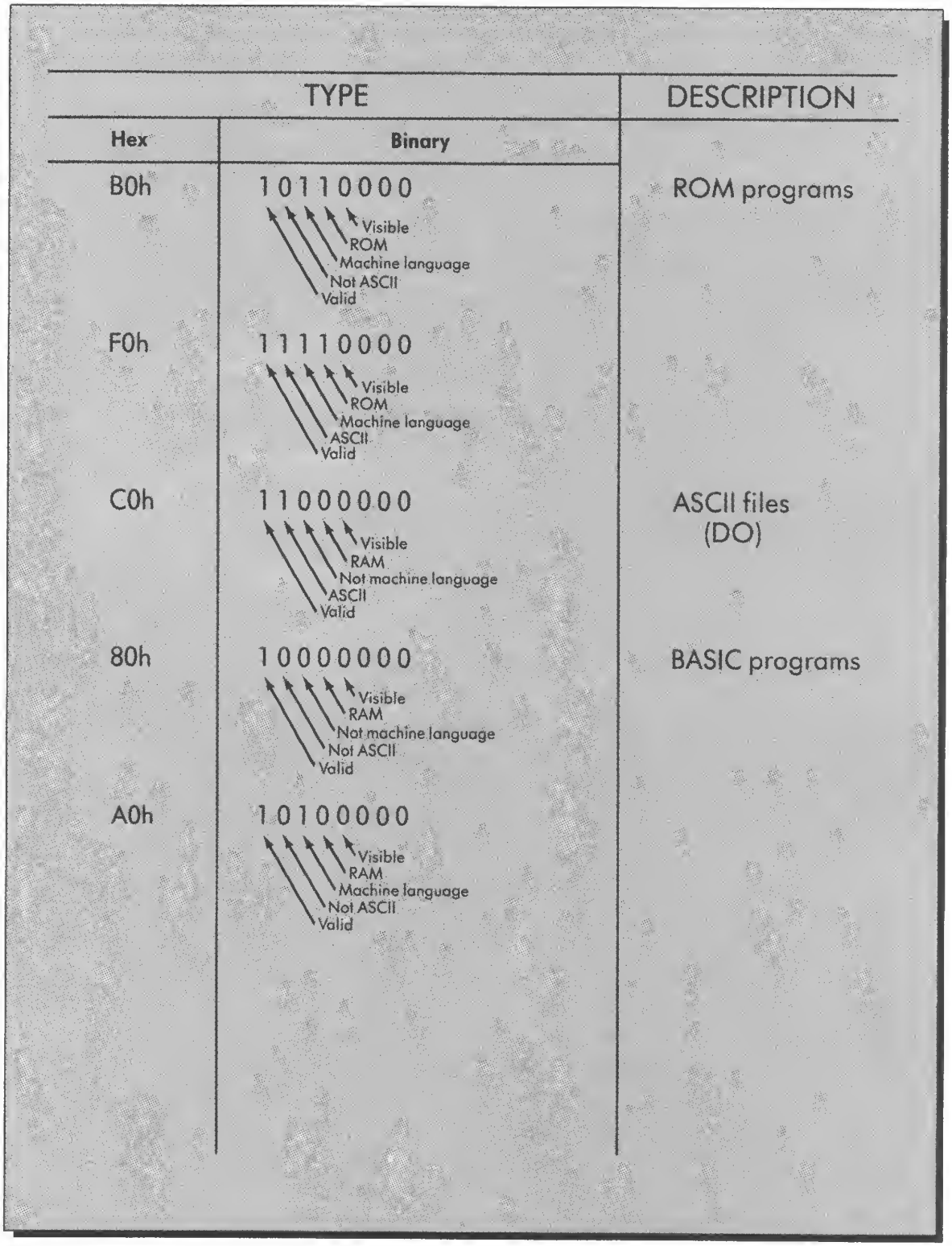


Figure 3-12. Displayed file types

comparing it against the specified file type in the C register. If the types do not match, it skips the entry. If they do, the filename is displayed at the appropriate place on the screen. A routine at 59C9h = 22,985d sets the proper cursor position on the screen for the entry.

The main directory display routine concludes by filling in missing entries with a "-.-" pattern and displaying "Select:" at the bottom of the screen.

The Command Loop

The main command loop extends from 585Ah = 22,618d to 588Bh = 22,667d. Its purpose is to get and interpret the user's keystroke commands for the MENU program.

The command loop starts with a conditional call to the BEEP routine (see Chapter 8), which is executed if the command buffer has overflowed. Next it calls a routine at 5D70h = 23,920d to update the time and date on the LCD screen (see box). Then it calls a routine at 5D64h = 23,908d to wait for a character from the keyboard. It checks for various special ASCII codes. An ASCII 13 (enter) causes it to jump to 58F7h = 22,775d, an ASCII 8 (backspace) or 7Fh = 127d (delete) causes it to jump to 588Eh = 22,670d, and an ASCII 15h = 21d (CTRL) U causes it to jump to 5837h = 22,583d. If the ASCII code is not one of these but is less than 20h = 32d, the loop jumps to 589Ch = 22,684d. If none of these occur, the character is printed as part of the command line on the bottom of the display.

Let's look at the cursor control commands in a bit more detail. The positions for the entries on the screen are numbered from 0 to 23 in the code that controls the cursor. The numbering system is simple: position 0 is the upper left position of the menu, position 1 is the position just to the right of position 0, and so on, left to right from top to bottom of the menu.

The current position is stored in FDEEh = 24,046d, and the current maximum position is stored at FDEFh = 65,007d. The right arrow routine increments the current position by 1, the left arrow routine decrements the current position by 1, the down arrow routine adds 4 to the current position, and the up arrow routine subtracts 4 from the current position. If the current position exceeds the number of entries or becomes negative, it is wrapped around. You can see how the cursor wraps or cycles around the display as you repeatedly hit any one arrow key.

The routine to handle (ENTER) actually does the dispatching to the selected menu entry. This routine is located at 58F7h = 22,775d. The location FDEDh = 65,005d is checked to see if the cursor position or the command line on the bottom of the screen should be used. If the cursor position is used, the routine counts through a table of addresses that maps the position numbers of the entries on the screen with the corresponding posi-

tions in the directory in RAM. This table begins at FDA1h = 64,929d. The ENTER routine finds the directory entry in RAM and checks the file type. For file type A0h = 160d, it jumps to 254Bh = 9547d; for file type B0h (ROM command files), it jumps to 596Fh = 22,895d; for file type F0h = 240d, it jumps to F624h = 63,012d (a RAM location); and for file type C0h = 192d (ASCII files), it jumps to 5F65h = 24,421d. If the file type is none of the above, it is treated as a BASIC file. After shoving the beginning address into location F67Ch = 63,100d and doing a couple of other things, the routine jumps to the execution loop of the BASIC interpreter.

The ADDRSS and SCHEDL Programs

The ADDRSS and SCHEDL programs are utilities that allow you to use your machine better. The ADDRSS program helps you to find addresses and telephone numbers that you have stored in your Model 100's memory, and the SCHEDL program helps you look up notes of things.

The code for the ADDRSS program extends from 5B68h = 23,400d to about 5B6Eh = 23,406d. The code for the SCHEDL program extends from 5B6Fh = 23,407d to about 5BA8h = 23,464d. Both programs jump to 5B74h = 23,412d.

Routine: ADDRSS

Purpose: To locate addresses and telephone numbers in the personal address directory file ADRS.DO

Entry Point: 5B68h = 23,400d

Input: None

Output: Enters the ADDRS program

BASIC Example:

```
CALL 23400
```

Special Comments: This CALL does not return to BASIC.

Routine: SCHEDL

Purpose: To locate schedule items in the personal notes file NOTE.DO

Entry Point: 5B6Fh = 23,407d

Input: None

Output: Enters the SCHEDL program

BASIC Example:

```
CALL 23407
```

Special Comments: This CALL does not return to BASIC.

The ADDRSS and SCHEDL programs behave very much like, and share much code with, the TEXT program described below. For this reason, we will not investigate these programs any further.

The TEXT Program

The TEXT program is the editor or word processor for the Model 100. The code for this program extends from 5DEEh = 24,046d to about 6BF0h = 27,632d. It consists of an initialization section, a main character input loop, and a set of routines to implement the various cursor control and editing commands.

Routine: TEXT

Purpose: To edit text files

Entry Point: 5DEEh = 24,046d

Input: None

Output: Enters the TEXT program

BASIC Example:

```
CALL 24046
```

Special Comments: This CALL does not return to BASIC.

The first part of the code for TEXT sets up the screen and gets the file to be edited. It calls a routine at 5A7Ch = 23,164d to set the function key display. It uses the table at 5E22h = 24,098d, which is empty. There are no function keys available at this point of the TEXT program. It then displays a message asking you to enter the file. It calls a routine at 2206h = 8710d to get the filename and locate the file and then jumps to 5F65h = 24,421d to edit the file. If a new file needs to be created, the routine called MAKTXT at 220Fh = 8719d is called. Location 5F65h = 24,421d is the same entry point dispatched to by the MENU program for ASCII files.

Routine: MAKTXT

Purpose: To create a text file

Entry Point: 220Fh = 8719d

Input: Upon entry, the filename must be stored in memory, starting at location FC93h = 64,659d. The .DO part of the file name need not be included.

Output: Enters the TEXT program

BASIC Example:

```
CALL 8719
```

Special Comments: None

The code at 5F65h = 24,421d sets up the screen for normal editing and loads the function keys with the options "Find", "Load", "Save", "Copy", and so on.

The main edit loop extends from 5FDDh = 24,541d to 6015h = 24,597d. The loop looks like a subroutine with a RET instruction at the end. However, at the top of the loop, the address of the top of the loop is pushed onto the stack. Thus the RET returns to the top each time.

At 5FEDh = 24,557d, the edit loop calls a routine at 63E5h = 25,573d to get the next key (see box). For control characters, it uses a table at 6015h = 24,597d to dispatch to routines to perform the various editing functions. For regular characters, it jumps to 608Ah = 24,714d, where it enters the character into the text.

Routine: Get Key

Purpose: To wait for a key for TEXT program

Entry Point: 63E5h = 25,573d

Input: From the keyboard

Output: Upon return, the ASCII code of the key is in the A register.

BASIC Example: Not applicable

Special Comments: Only called from TEXT

The Initialization Routines

The initialization routines help set up or configure the various I/O devices in the Model 100 to start the computer after it gets stuck and will not respond to you through the keyboard.

The code for warm and cold I/O initialization runs from about 6CD6h = 27,862d to 6D3Eh = 27,966d (see boxes). The last part of the ROM, starting at 7D33h = 32,051d, contains code to start up the computer.

Routine: INITIO — Cold Start

Purpose: To cold start the I/O of the Model 100

Entry Point: 6CD6h = 27,862d

Input: None

Output: The Model 100 I/O is reset — cold start.

BASIC Example:

```
CALL 27862
```

Special Comments: Watch out, this is a cold restart! It clears the area from FF40h = 65,344d to FFFDh = 65,533d. This area contains the keyboard buffer, among other things. It continues into the warm start reset after clearing that area.

Routine: INITIO — Warm Start

Purpose: To warm start the I/O of the Model 100

Entry Point: 6CE0h = 27,872d

Input: None

Output: Initializes the I/O devices of the Model 100.

BASIC Example:

```
CALL 27872
```

Special Comments: None

The Primitive Device Routines

The code for the primitive-level routines for handling devices runs from about 6D3Fh = 27,967d to 7D32h = 32,050d and includes tables. In subsequent chapters we will study many of these routines in detail.

Summary

In this chapter we have surveyed the Model 100's ROM from beginning to end. We have concentrated on the areas that manage the BASIC interpreter and the MENU program, for these are the keys to understanding many other secrets of the Model 100's operation.

A particular area of interest is the code for the LET command, in which the BASIC interpreter finds variables and evaluates expressions. We have shown how you can gain direct control of this code to make an interactive function evaluator.

4

Hidden Powers of the Liquid Crystal Display

Concepts

How liquid crystal displays work
How to program the LCD

The Liquid Crystal Display (LCD) is the main output device for the Model 100. As such, it provides a good starting point for understanding the operation of the Model 100. The LCD also represents a new approach in display technology, an approach that has much promise because it requires less power and space than the older video technology. It is one of the major reasons why the Model 100 is truly portable.

We will start our exploration of the LCD with a general description of liquid crystal displays and then see in detail how the built-in display of the Model 100 works. We will see how to program the display screen, both directly and by calling various levels of subroutines in the computer's ROM.

How Liquid Crystal Displays Work

In contrast to the more traditional video CRT (cathode ray tube), a liquid crystal display does not generate its own light. Instead, it selectively blocks light that comes from the outside.

If you look closely at the LCD of your Model 100, you will see that it consists of a two-dimensional array of tiny squares. These are the picture elements (pixels) of the display (see Figure 4-1). The horizontal pixel positions are numbered from 0 to 239 from left to right, and the vertical positions are labeled from 0 to 63 from top to bottom.

Each tiny square is a sandwich in which the "bread" consists of polarized filter material and the "filling" is made of liquid crystal. Glass plates separate the liquid crystal from the polarizing filters in this sandwich (see Figure 4-2).

Each pixel can be individually lightened or darkened by applying a voltage to it that affects its transparency. A layer of reflective material behind the entire display helps bounce the light through those pixels of the display that are transparent.

To understand how the pixels can be made more or less transparent, you must understand a little about the theory of light. Light is electromagnetic radiation; that is, it consists of combinations of electrical and magnetic waves.

Ordinary light consists of a hodgepodge of individual light waves. Each individual light wave is a precisely balanced pair of electrical and magnetic waves. These component waves both travel in the same direction, the direction of motion of the wave; they vibrate at the same frequency, the frequency of the wave; but their directions of vibration are perpendicular to each other and to the direction of the motion of the wave (see Figure 4-3). The two directions of vibration determine an oriented plane called the "plane of vibration". This gives an orientation to each individual light wave.

A polarizing filter polarizes light by allowing only those light waves to pass through it that are oriented in a certain way. That is, it blocks light

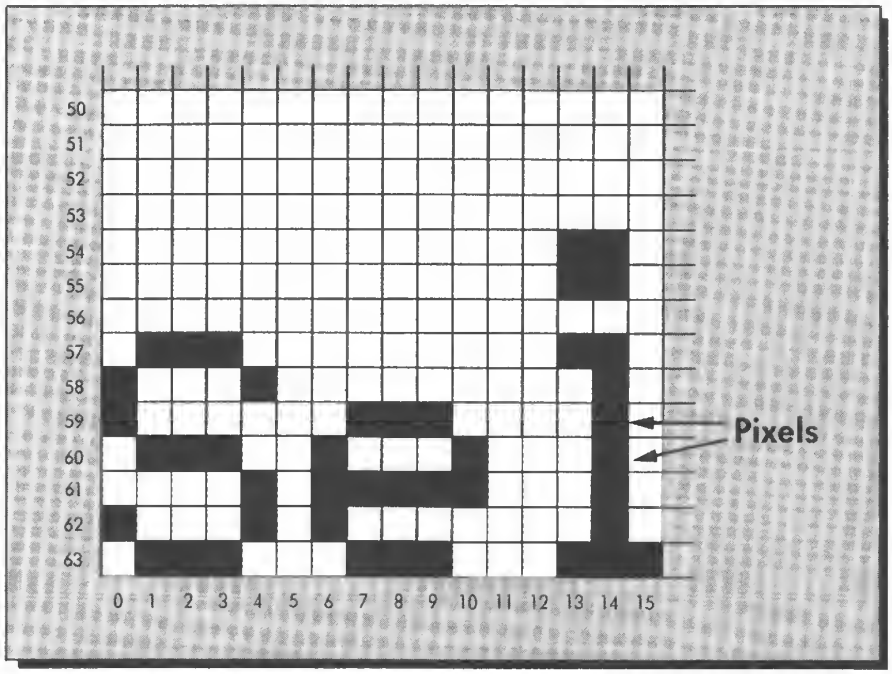


Figure 4-1. Pixels of LCD display

waves whose electrical (or magnetic) components do not line up in a certain direction. Light that has passed through such a filter is said to be "polarized", because the orientations of its individual light waves are closely aligned with each other.

When two polarizing filters are placed together face to face, they will let through light if their polarizations are aligned but will block most of the light if their polarizations are twisted with respect to each other.

Liquid crystal twists the orientation of the light that passes through it. The amount of the twist depends upon the voltage applied to the crystal. When the crystal is sandwiched between two polarizing filters, this twisting, and hence the voltage applied to the liquid crystal, is translated into the degree of transparency of the sandwich.

Light comes into an LCD display from the outside, goes through the sandwich, is reflected from the mirrorlike surface behind the display, and comes back through the display to your eyes. The amount of light that makes its way through this arrangement depends on your viewing angle as well as the voltage that is applied to the liquid crystal. The adjustment wheel on the right side of your Model 100 allows you to select the best voltage for optimum visibility from your particular viewing angle.

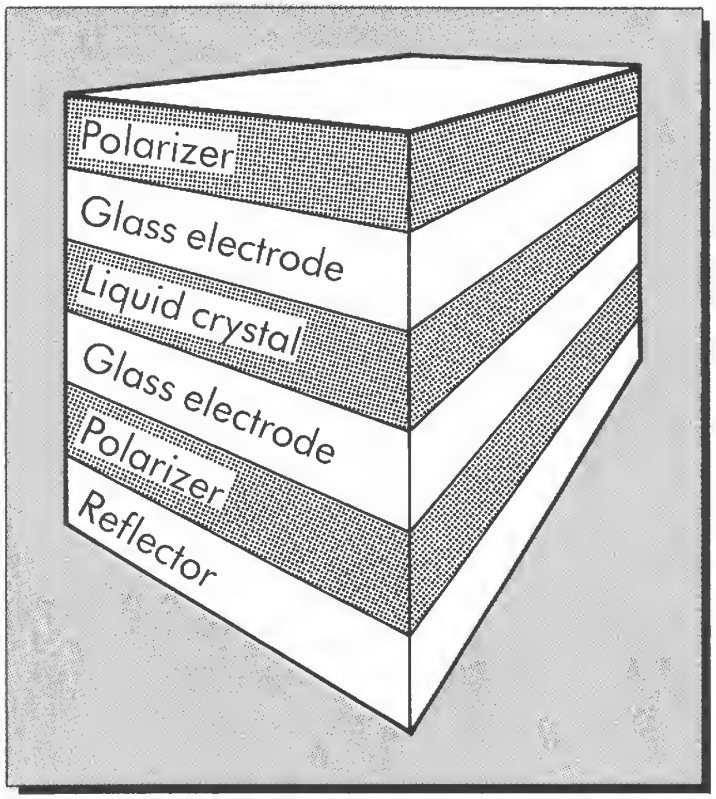


Figure 4-2. Pixel sandwich

Each pixel of the display can be individually controlled with its own voltage. On the Model 100, a display element is transparent when little voltage is applied and becomes opaque as more voltage is applied. On some systems direct current is used, but on the Model 100 alternating current is used to extend the life of the display.

The Model 100's display screen consists of a 240 by 64 array of pixels. Ten chips called "LCD horizontal drivers" directly control ten different regions of the display. Each horizontal driver has 50 lines that can control 50 horizontal positions of the display. The drivers come in pairs, one to control the upper 32 rows and one to control the lower 32 rows of each horizontal section of the display (see Figure 4-4). You can see from this figure that four pairs of horizontal LCD drivers control the first 200 horizontal positions (in 50-position sections) and one pair controls the last 40 positions. These last two horizontal drivers have only 40 of their 50 outputs connected to the display.

Each horizontal LCD driver stores a total of 1600 bits, one for each of the pixels in a 50 by 32 section of the display (except of course for the last pair of drivers, which don't map to a complete 50 by 32 section). The 1600

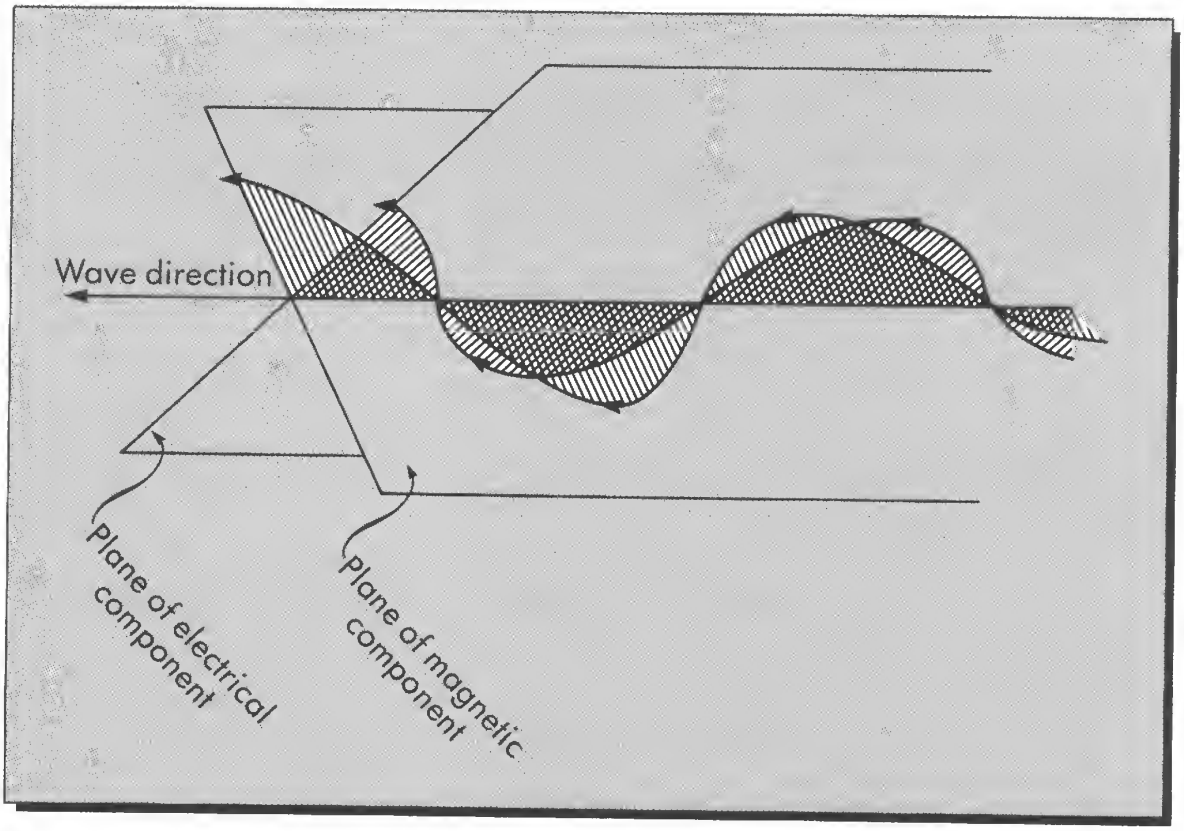


Figure 4-3. Components of a light wave

bits are stored in four banks of 50 bytes. Each bank corresponds to a 50 by 8 strip of the display. Bank 0 corresponds to the top eight rows of pixels, bank 1 corresponds to the next eight rows, bank 2 corresponds to the next eight rows, and the last eight rows correspond to bank 3. Within each bank, each byte corresponds to a 1 by 8 column of pixels (see Figure 4-5).

The horizontal LCD drivers continually refresh the display on their particular sections of the screen. Each output line to the display refreshes a 1 by 32 column of pixels. The information is sent through these lines serially, first the top row, then the second row, and so on, over and over again, creating a top-to-bottom scanning pattern.

A pair of vertical LCD drivers controls the rows of the display, enabling and disabling them in synchronization with the above-mentioned scanning pattern. One vertical driver controls the upper 32 rows of pixels, and the other controls the lower 32 rows. The two vertical drivers scan at the same rate and time through their own parts of the display. When the horizontal drivers produce the information for their first row, the vertical drivers enable only their first row, and so on (see figure 4-6).

The scanning rate is one row about every 446 microseconds. The entire 32 rows are scanned about every 14.3 milliseconds, or 70 times a second. This is slightly faster than a CRT display is normally scanned.

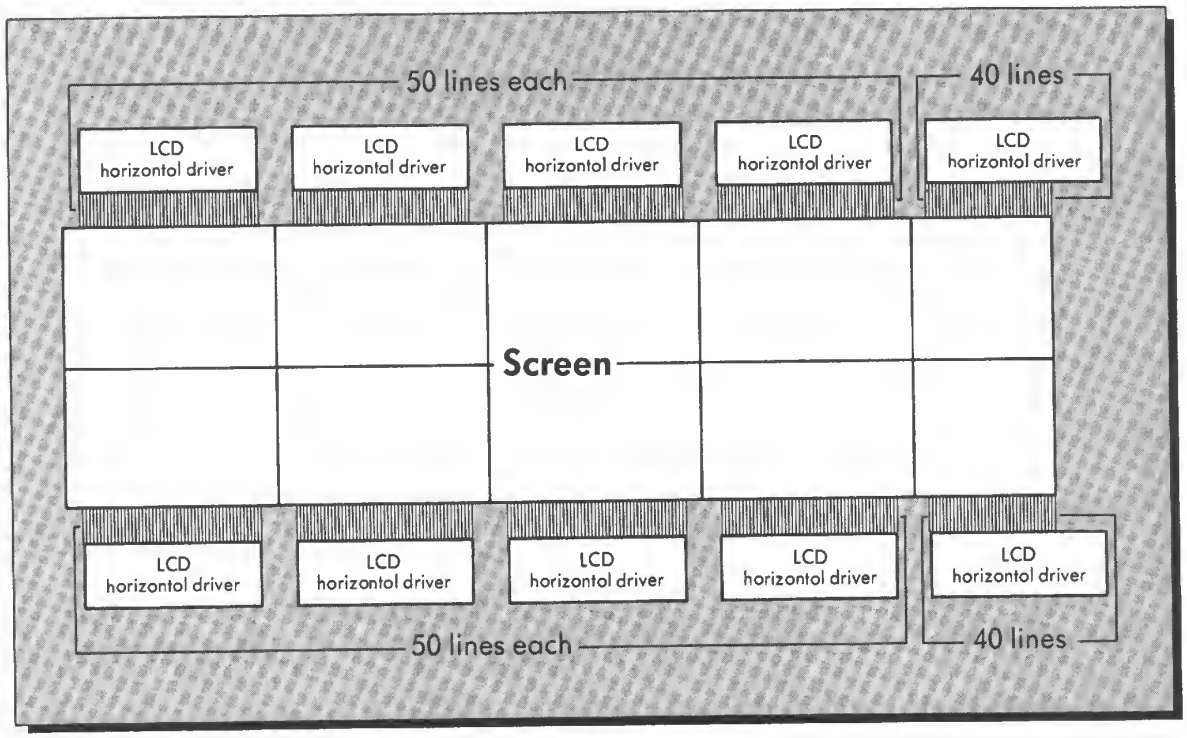


Figure 4-4. Horizontal LCD drivers

In contrast to the horizontal drivers, the vertical drivers do not store any information; they merely maintain a constant scanning pattern.

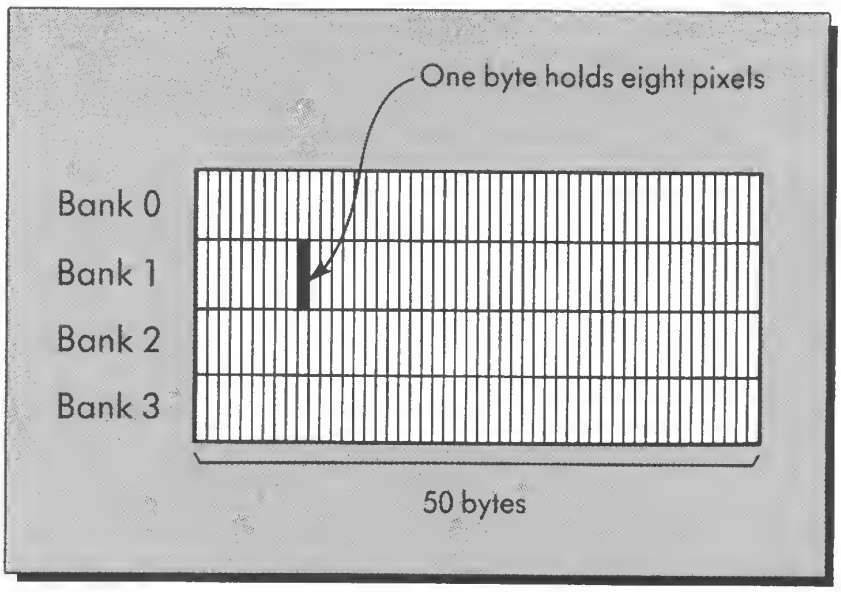


Figure 4-5. Banks and bytes in a horizontal LCD driver

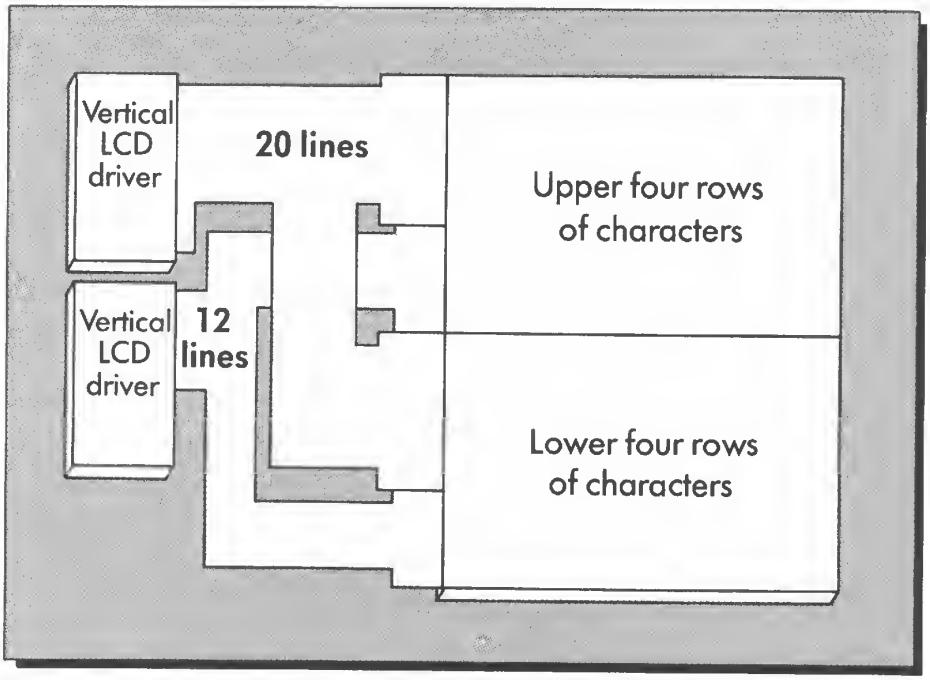


Figure 4-6. Vertical LCD drivers

How to Program the LCD

You can program the LCD screen by sending bytes to the horizontal LCD driver chips. These bytes can also be read back at a later time.

You can separately address each individual pixel in the entire display. To do this, you must determine the pixel's LCD driver, its "bank" within the LCD driver, its horizontal byte position within the bank, and its position within that byte. The Model 100 figures this out each time you ask it to plot a point with the PSET or PRESET command, and it makes a similar computation each time it puts a character on the screen. In this chapter we will see exactly how this works.

To control the LCD, you use port FEh = 254d to send commands and read status and port FFh = 255d to send and receive data bytes (see Figure 4-7). We will discuss this in more detail later.

Ports B9h = 185d and BAh = 186d specify which of the ten horizontal drivers is being addressed. For port B9h = 185d, bits 0 through 4 control the selection of the five LCD drivers across the top of the display in left to right order, and bits 5 through 7 control three of the LCD drivers on the lower left part of the display. Bits 0 and 1 of port BAh = 186d control the remaining two LCD drivers for the lower right part of the screen (see Figure 4-7). To turn an LCD driver "on", so that it can receive a command or transfer data, put a one in the corresponding bit; and to turn it "off", put a zero in that bit. For example, if you put 00001010 binary into port B9h = 185d and the binary pattern 10 into bits 0 and 1 of port BAh = 186d, then the LCD drivers will be "on" and "off" in the following pattern:

off on off on off

off off off off on

Usually, only one LCD driver is programmed at a time; thus there is usually only one "1" bit, with the rest equal to "0".

Several words of warning are needed about using ports B9h = 185d and BAh = 186d because they are also used for a number of other functions, including the power, keyboard, clock, buzzer, and communications lines. Since keyboard scanning, cursor blinking, and clock reading are going on constantly as a background task, you must turn this background task off before selecting the LCD drivers. This is somewhat dangerous, since if you don't turn this task back on, your keyboard will no longer work, and you will lose control of your machine. Fortunately, when you program in BASIC, the keyboard-cursor-clock is turned back on for the INPUT statement and as BASIC finishes running your program.

You should be careful not to disturb bits 2 through 7 of port BAh = 186d. In fact, if you put a zero into bit 4 of this port, you will turn off the power to the machine! To properly program this port, you must read the port first, AND its contents with the mask 11111100 binary, OR the contents with 000000aa binary, where aa is the desired pattern for bits 0 and 1, and then put the result back into the port.

LCD commands, which are sent through port FEh = 254d, allow you to turn the display on and off and specify how data bytes are to be loaded in and out of the LCD driver chip. Each command byte consists of two parts or fields: a two-bit bank selector field that is stored in bits 6 and 7, and a six-bit field that is stored in bits 0 through 5.

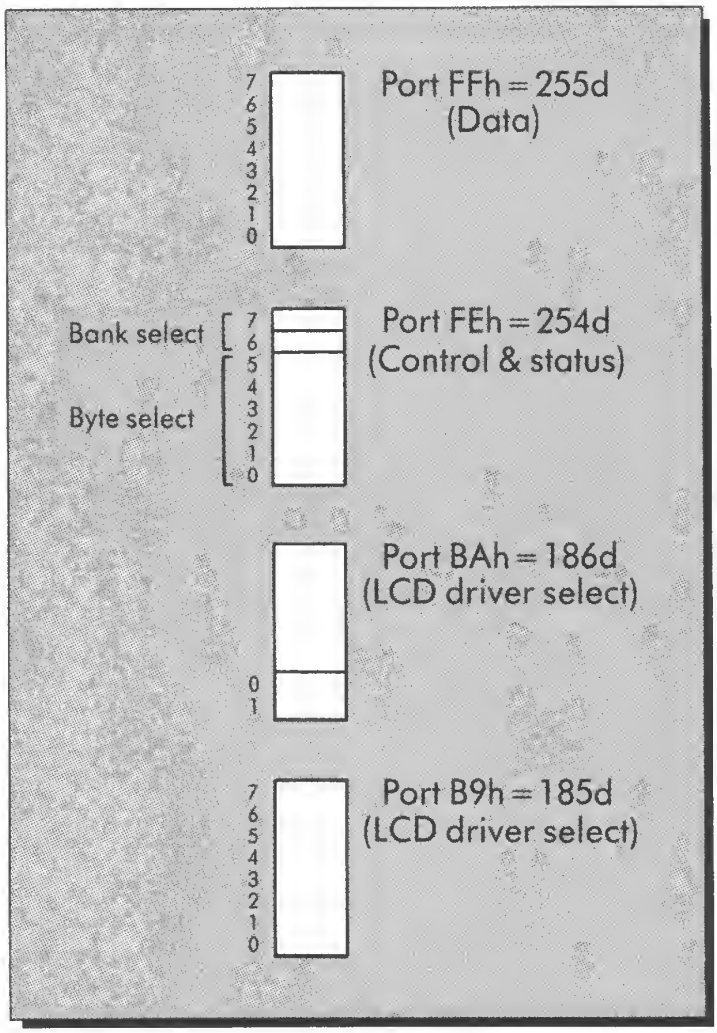


Figure 4-7. Control, status, and data ports for the LCD

The six-bit field of the command byte contains a number between 0 and 63. If this number is in the range from 0 to 49, it indicates a horizontal byte position in the bank specified by the two-bit bank select field. This byte position is called the current byte position and is used to indicate where the next byte will be loaded in or out of the driver. For example, if you send the command byte the binary value 01000011, then the bank select field is 01, and the horizontal position field is 000011 binary or 3 decimal. Thus the next data byte will be at position 3 of bank 1 (see Figure 4-8).

Values greater than 49 in the six-bit field program the LCD driver in other ways. For example, a value of 56 in the six-bit field turns off the display, making the corresponding part of the screen blank; a value of 57 turns the display back on, and values of 58 and 59 affect the order in which bytes are to be loaded in or out of the display.

After a data byte is loaded in or out of the driver (through port FFh = 255d), the current position is advanced. Normally the position advances to the right, but you can reverse the direction by sending a command byte with a value of 58 in its six-bit field. In this mode bytes are loaded into the LCD in right-to-left order. To return to the normal left-to-right loading order, send a command byte of 59.

As we noted above, to address an individual pixel of the display, you must determine its LCD driver, its bank, its horizontal byte position, and its position within that byte. Here is a BASIC program that illustrates how these considerations can be used to plot a pixel anywhere on the screen.

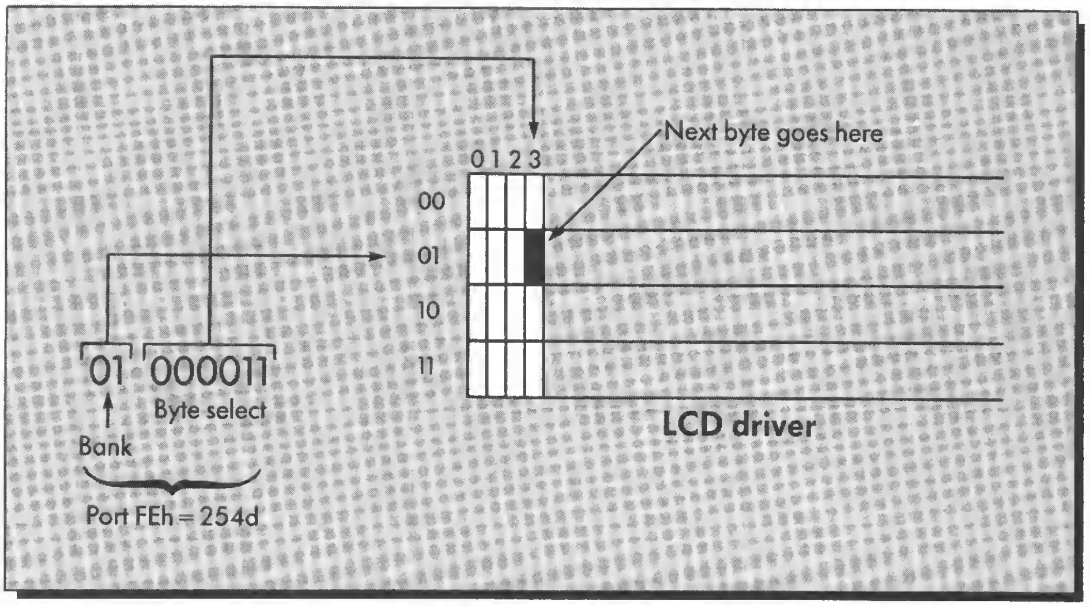


Figure 4-8. Loading an LCD driver

```
100 ' LCD DIRECT PROGRAMMING
110 CLS
120 INPUT "LCD ENABLE BITS (0-1023)";E
130 INPUT "LCD BANK (0-3)";B
140 INPUT "LCD BYTE POSITION (0-63)";H
150 INPUT "LCD BYTE VALUE (0-255)";V
160 INPUT "NUMBER OF BYTES (>0)";N
170 CALL 30300
180 OUT 185, E AND 255
190 P1 = INP(186) AND 254
200 OUT 186, P1 OR (3 AND E/256)
210 OUT 254, 64*(B AND 3) + (63 AND H)
220 FOR I = 1 TO N
230  OUT 255, 255 AND V
240 NEXT I
```

Line 110 clears the screen so that you can better see the program's result. It also forces the input statement to the top of the screen so that the display won't scroll and make your result disappear before you can examine it. Lines 120-160 input five parameters, specifying the acceptable ranges for their values (see Figure 4-9). E is the ten-bit enable/disable pattern that is sent to ports B9h = 185d and BAh = 186d. This selects a combination of LCD drivers. B selects one of the four banks in the selected LCD drivers. H selects the horizontal byte position within the bank. V specifies the bit pattern for the byte. N specifies the number of bytes that will be sent to the LCD drivers. Line 170 turns off the clock-cursor-keyboard background task. (This routine is located at 765Ch = 30,300d, as we shall see later.) Line 180 sets the lower eight enable/disable bits, and lines 190-200 set the upper two enable/disable bits for the LCD drivers. Line 210 computes the bank and horizontal position command byte and sends it out port FEh = 254d. Lines 220-240 form a FOR...NEXT loop to send the data byte out port FFh = 255d the specified number of times. Try typing this program in and running it.

In the next section we will see how the Model 100 controls these quantities to plot points, lines, boxes, and characters.

ROM Routines for the LCD

The Model 100 ROM contains routines to plot and erase points, to draw lines and boxes, and to print characters on the screen. It also contains code to make the cursor blink. We'll discuss these routines in detail so you can learn how to take advantage of them. You will find them useful for creating special effects such as scrolling subsections of the screen and making real-time displays of complex data. Having complete control of the screen is especially useful if you are designing games or educational programs.

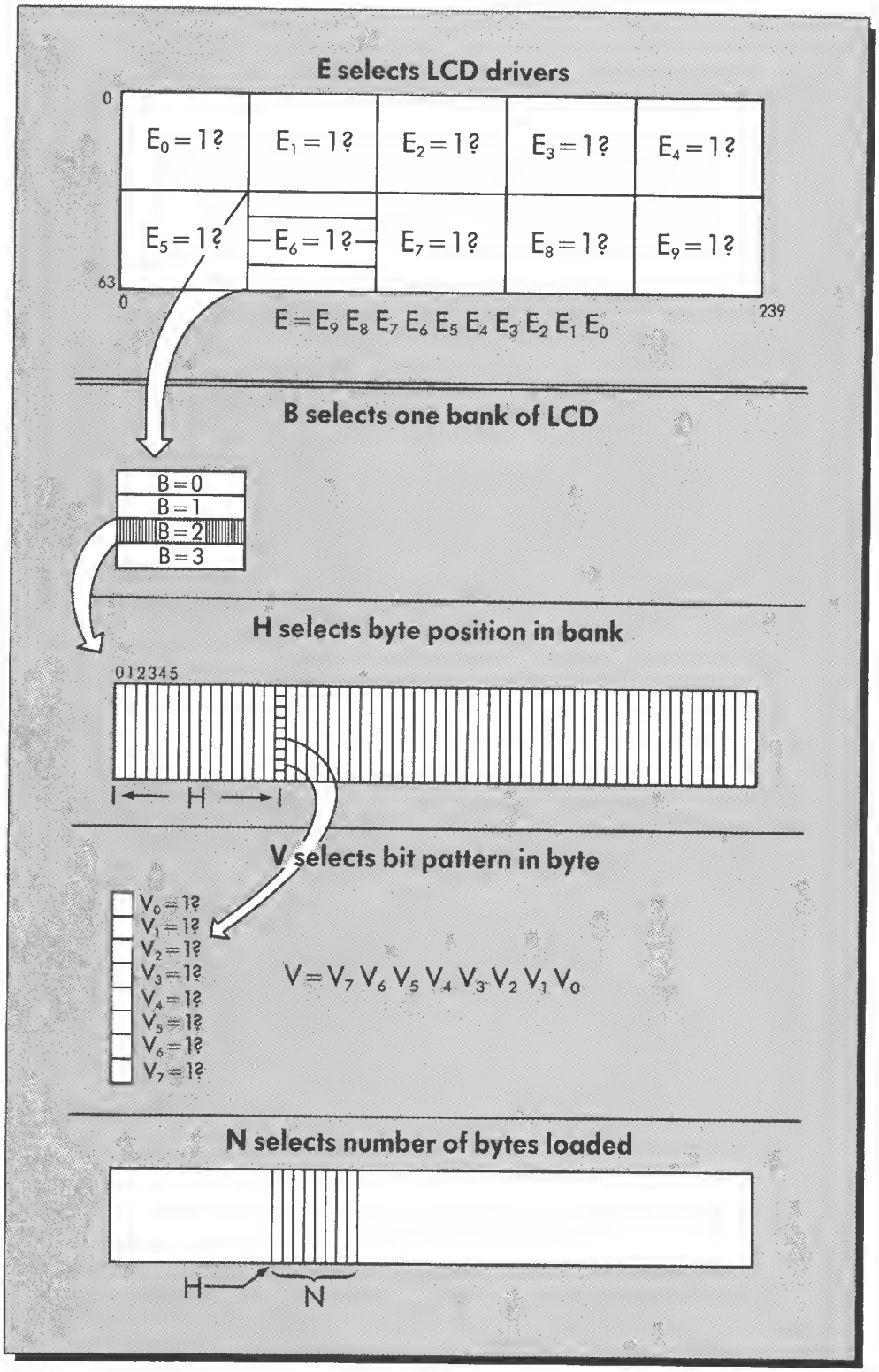


Figure 4-9. How E, B, H, V, and N program the LCD

Point Plotting

Let's start with the point-plotting routines. They give you control of each individual dot on the screen. In BASIC, the PSET and PRESET commands are used to plot points. PSET is used to turn on pixels, and PRESET is used to turn them off.

PSET and PRESET

BASIC commands are implemented as routines in the ROM. The routine for PSET starts at 1C57h = 7255d (see box), and the routine for PRESET starts at 1C66h = 7270d (see box). These routines first call a routine starting at 1D2Eh = 7470d, which gets the (x,y) coordinates of the point from the BASIC command line (see box).

Routine: PSET Command

Purpose: To plot a point on the LCD screen

Entry Point: 1C57h = 7255d

Input: Upon entry, the HL register pair points to the end of the PSET command line, which contains the coordinates of the point in tokenized form. (See the *TRS-80® Model 100 Portable Computer* manual for the syntax of the PSET command.)

Output: To the screen

BASIC Example:

```
CALL 7255,0,63105
```

where the input buffer at F681h = 63,105d contains a tokenized BASIC PSET command line starting with the coordinates of the point. Call the tokenizer routine at 646h = 1606d before using this example.

Special Comments: The input buffer at F681h = 63,105d is also used by the INPUT command.

Routine: PRESET Command

Purpose: To erase a point on the LCD screen

Entry Point: 1C66h = 7270d

Input: Upon entry, the HL register pair points to the end of the PRESET command line, which contains the coordinates of the point in tokenized form. (See the *TRS-80® Model 100 Portable Computer* manual for the syntax of the PRESET command.)

Output: To the screen

BASIC Example:

```
CALL 7270,0,63105
```

where the input buffer at F681h = 63,105d contains a tokenized BASIC PRESET command line starting with the coordinates of the point. Call the tokenizer routine at 646h = 1606d before using this example.

Special Comments: The input buffer at F681h = 63,105d is also used by the INPUT command.

Routine: Get (x,y) Coordinate

Purpose: To get (x,y) coordinate from BASIC command line

Entry Point: 1D2Eh = 7470d

Input: Upon entry, the HL register pair points to a tokenized string containing the (x,y) coordinates.

Output: When the routine returns, the D register contains the value of the x-coordinate and the E register contains the value of the y-coordinate.

BASIC Example: Not directly applicable

Special Comments: None

PLOT/UNPLOT

For PSET the plotting routine is at 744Ch = 29,772d and is called PLOT (see box), and for PRESET it is at 744Dh = 29,773d and is called UNPLOT (see box). These are really two entries into the same plotting routine. In the first case, a nonzero value is placed in the A register at the beginning of the routine. In the second case, the A register is cleared at the beginning of the plotting routine. In either case, the x-coordinate (horizontal position) is in the D register, and the y-coordinate (vertical position) is in the E register.

Routine: PLOT

Purpose: To plot a point on the LCD display screen

Entry Point: 744Ch = 29,772d

Input: Upon entry, the D register contains the x-coordinate and the E register contains the y-coordinate.

Output: To the screen

BASIC Example: Not directly applicable

Special Comments: None

Routine: UNPLOT

Purpose: To erase a point on the LCD display screen

Entry Point: 744Dh = 29,773d

Input: Upon entry, the D register contains the x-coordinate and the E register contains the y-coordinate.

Output: To the screen

BASIC Example: Not directly applicable

Special Comments: None

The first action taken by the PLOT/UNPLOT routine is to call a routine starting at 765Ch = 30,300d that turns off the clock-cursor-keyboard background task (see box). Specifically, this routine turns off interrupt number 7.5, which is normally generated by the clock chip every four milliseconds to run the clock-cursor-keyboard background task. (This interrupt will be discussed in more detail in Chapters 5 and 6.)

Routine: Turn Off and Reset Interrupt 7.5

Purpose: To turn off and rearm interrupt 7.5

Entry Point: 765Ch = 30,300d

Input: None

Output: When the routine returns, interrupt 7.5 is disabled and rearmed for the next time it is enabled.

BASIC Example:

```
CALL 30300
```

Special Comments: The interrupt can be reenabled in a number of ways, such as by means of a PRINT command or the termination of a BASIC program.

The PLOT/UNPLOT routine divides the x-coordinate by 50. The quotient determines which pair of horizontal LCD drivers controls the pixel, and the remainder determines the byte position within the driver.

Enabling the LCD Drivers

The y-coordinate is processed to determine whether the pixel is in the upper or lower half of the screen. This determines which set of five horizontal LCD drivers should be addressed. The HL register points to the first half of a table in memory for the upper half of the screen and the second half of the table for the lower half of the screen (see Figure 4-10). This table contains the bit patterns for enabling the LCD drivers through ports B9h = 185d and BAh = 186d. The quotient determined by the PLOT/UNPLOT routine is added to the HL register to point to the bit pattern for the desired LCD driver. Then the routine at 753Bh = 30,011d (see box) is called to send these enable/disable bits to select the correct LCD driver.

Routine: Enable LCD Drivers

Purpose: To enable LCD drivers

Entry Point: 753Bh = 30,011d

Input: Upon entry, the HL register pair points to an entry in special tables in memory that contain bit patterns to set the 8155 PIO chip that controls the LCD drivers. There are two such tables. One table begins at 7551h = 30,033d, has three bytes per entry, and is indexed by the column position for character positions. The other (see Figure 4-10) begins at 7643h = 30275d, has two bytes per entry, and is indexed by the particular LCD driver.

Output: The specified LCD driver is enabled, and the others are disabled.

BASIC Example:

```
CALL 30011,0,30275+2*L
```

where L is a number between 0 and 9 and indicates the particular LCD driver.

Special Comments: None

The bits that determine the bank number are shifted into the upper two bits, their correct position within the command byte. The bank bits and the horizontal byte position are combined and stored in the B register, ready to be sent to the command port of the LCD driver.

Because the bit for the pixel is stored within a byte that has bits for seven other pixels, the contents of the byte must be read first. This way the values for the other bits can be preserved when the byte is put back.

The routine to read the byte from the LCD driver is located at 74F5h = 29,941d (see box). It waits for the LCD driver status to indicate that the driver is ready to receive a command. Then it sends the command byte. Finally, it reads the data byte (when status indicates the data byte is ready). The ready status is contained in bit 7 of port FEh = 254d.

Routine: Read LCD Bytes

Purpose: To read a sequence of bytes from an LCD driver

Entry Point: 74F5h = 29,941d

Input: Upon entry, the B register contains a command byte for the selected LCD driver, the HL register pair points to an area of memory to which the bytes are to be transferred, and the E register contains the number of bytes to be transferred. The command byte usually selects the bank and byte position within the byte. See the text for further explanation.

Output: When the routine returns, the bytes from the LCD driver are in memory, starting at the location specified by HL upon entry.

BASIC Example: Not directly applicable

Special Comments: None

Address	Binary data		Address
	76543210	76543210	
7644h	00000000	00000001	7643h
7646h	00000000	00000010	7645h
7648h	00000000	00000100	7647h
764Ah	00000000	00001000	7649h
764Ch	00000000	00010000	764Bh
764Eh	00000000	00100000	764Dh
7650h	00000000	01000000	764Fh
7652h	00000000	10000000	7651h
7654h	00000001	00000000	7653h
7656h	00000010	00000000	7655h
	Sent to port BAh	Sent to port B9h	

Figure 4-10. Table for enabling LCD drivers for point plotting

The y-coordinate is used to determine a mask for the bit position within the byte. This uses some modular arithmetic and the same table that was used for the ports B9h = 185d and BAh = 186d. The mask contains a "1" in the correct bit position and "0" in the other positions. For the PSET command, the mask is ORed with the byte just read from the LCD driver. For the PRESET command, the mask is used to clear the appropriate bit of this byte.

The correctly modified byte is sent back to the LCD driver by calling the routine at 74F6h = 29,941d (see box). This works almost the same as the read routine; indeed, it is the same except for a byte at its entry.

Routine: Write LCD Bytes

Purpose: To write a sequence of bytes to an LCD driver

Entry Point: 74F6h = 29,914d

Input: Upon entry, the B register contains a command byte for the selected LCD driver, the HL register pair points to an area of memory from which the bytes are to be transferred, and the E register contains the number of bytes to be transferred. The command byte usually selects the bank and byte position within the byte. See the text for further explanation.

Output: When the routine returns, the bytes from the specified memory area are transferred to the LCD driver.

BASIC Example: Not directly applicable

Special Comments: None

The last thing that the PLOT/UNPLOT routine does is turn on the 7.5 interrupt so that the clock-cursor-keyboard background task can continue. The routine to do this is located at 743Ch = 29,756d (see box). Remember that this background task must continue to operate in order for the keyboard to function.

Routine: Turn on Interrupt 7.5

Purpose: To enable interrupt 7.5

Entry Point: 743Ch = 29,756d

Input: None

Output: To the interrupt control

BASIC Example:

```
CALL 29756
```

Special Comments: None

Line Drawing

Line drawing and area filling are at the next higher level above point plotting. The routine to draw lines starts at 1C6Dh = 7277d (see box). It can be invoked from the BASIC LINE command. The LINE command has a number of parameters and options. It can draw lines and rectangles (filled or unfilled). The endpoints of the lines and the corners of the rectangles can be specified either as a pair of points in the form

(x1,y1)-(x2,y2)

or as a single point in the form

-(x2,y2)

You can see that in the second case, the first point is not specified. Instead, an unseen graphic cursor called the "CP" (current position) is used. Each time a point, line, or box is drawn with the PSET, PRESET, or LINE commands, the CP is updated to the last point referenced. This is done at 1D46h = 7494d in the routine by getting the (x,y) coordinates from the command line. The CP is stored at location F64Eh = 63,054d.

Routine: LINE — BASIC command (Graphics)

Purpose: To draw a line on the LCD screen

Entry Point: 1C6Dh = 7277d

Input: Upon entry, the HL register pair points to the end of the LINE command line, which contains the coordinates of the point in tokenized form. See the *TRS-80® Model 100 Portable Computer* manual for the syntax of the LINE command.

Output: To the screen

BASIC Example:

```
CALL 7277,0,63105
```

where the input buffer at F681h = 63,105d contains a tokenized BASIC LINE command line starting with the coordinates of the point. Call the tokenizer routine at 646h = 1606d before using this example.

Special Comments: None

The line-drawing routine uses a form of Bresenham's line-drawing algorithm and calls either PLOT or UNPLOT to plot or erase points along the line, depending upon the particular "color" chosen. Bresenham's line drawing algorithm is a well-known method for drawing lines quickly. It consists of an initialization stage and a tight loop that steps through the pixels along the line, performing a series of vertical, horizontal, and diagonal moves. In the Model 100's ROM the initialization part starts at 1CD9h = 7385d, and the loop starts at 1D0Ch = 7436d.

The box-drawing option of the LINE command calls the Bresenham algorithm four times, once for each side of the box. The routine for this is at 1CBCh = 7356d (see box). A box-fill option at 1CA5h = 7333d (see box) has a loop that calls the Bresenham algorithm over and over again to fill in all the rows inside a specified rectangle.

Routine: Box — Unfilled

Purpose: To draw an unfilled box on the LCD screen

Entry Point: 1CBCh = 7356d

Input: Upon entry, the coordinates of two opposite corner points of the box are on the stack. For each of the two corner points there is one word on the stack. The upper byte of this word contains the x-coordinate, and the lower byte contains the y-coordinate.

Output: To the screen

BASIC Example: Not directly applicable

Special Comments: None

Routine: Box — Filled

Purpose: To draw a filled box on the LCD screen

Entry Point: 1CA5h = 7333d

Input: Upon entry, the coordinates of two opposite corner points of the box are on the stack. For each of the two corner points there is one word on the stack. The upper byte of this word contains the x-coordinate, and the lower byte contains the y-coordinate.

Output: To the screen

BASIC Example: Not directly applicable

Special Comments: None

Character Plotting (Text)

The routines to plot characters on the screen are more complicated than the point- or line-drawing routines. The text routines have a multitude of levels; that is, a higher-level routine calls a lower-level routine, which calls a still lower-level routine, and so forth. We'll go through these routines level by level.

Level 1

Character plotting at the highest level (level 1) can be called by the RST 4 instruction. This is a software interrupt; that is, it is a CPU instruction that acts just like a hardware interrupt. The RST 4 instruction calls whatever routine is located at address 20h = 32d. In the Model 100's ROM, address 20h = 32d is the start of a jump to 4B44h = 19,268d, which is where the character output routine is actually located.

The level 1 character-plotting routine at 4B44h = 19,268d displays a character on the screen at the current cursor position (see box). Before the routine is called, the ASCII code must be in the A register. This routine is called LCD by Radio Shack, but it can be used to direct output to other devices as well, such as the printer and the optional CRT display screen. Perhaps a better name for this routine would be CONSOLE OUT. You can access this routine directly by typing:

```
CALL 32,65
```

or

```
CALL 19268,65
```

In both cases this will place an uppercase "A" on the screen.

Routine: Character Plotting — Level 1

Purpose: To print a character on the LCD screen

Entry Point: 4B44h = 19,268d

Input: Upon entry, the A register contains the ASCII code of the character to be printed.

Output: To the screen

BASIC Example:

```
CALL 19268,A
```

where A is the ASCII code of the character to be printed.

Special Comments: RST 4 also calls this routine (see Chapter 3).

Location F675h = 63,093d contains what we call the "print flag". The level 1 routine checks to see if the print flag is nonzero. If it is nonzero, output from this routine goes to the printer. If it is 0, the routine branches to 4BAAh = 19,370d, where the level 2 character plotting routine is called.

Level 2

The level 2 character-plotting routine is at 4313h = 17,171d (see box). It has "hooks" to allow user-defined routines to be called (see Figure 4-11). By "hook" we simply mean a way for users to attach their own routines.

Routine: Character Plotting — Level 2

Purpose: To print a character on the LCD screen

Entry Point: 4313h = 17,171d

Input: Upon entry, the ASCII code of the character to be printed is in the A register.

Output: To the screen

BASIC Example:

```
CALL 17171,A
```

where A is the ASCII code of the character to be plotted.

Special Comments: None

These "hooks" are done through the RST 7 interrupt instruction (see box in Chapter 3). This instruction calls location 38h = 56d, which jumps to location 7FD6h = 32,726d, where there is a dispatcher routine. The dispatcher routine calls one of the routines whose addresses are stored in a big table in RAM called the "hook" table. This table starts at address FADAh = 64,218d. Normally, the addresses in the first half of the hook table all point to a routine that consists of just a return instruction. The addresses in the second half of this table point to a routine that outputs the illegal function error message. The byte following the RST 7 instruction is used to index into the hook table. For level 2 character plotting, this index is 8. This points to a return instruction. If you want to use your own routine, put its address into the RAM address FADAh + 8, and it will be called every time you plot a character on the screen.

Level 3

After the level 2 character-plotting routine calls the "hook", it calls the level 3 character-plotting routine.

The level 3 character-plotting routine at 431Fh = 17,183d checks location F638h = 63,032d (see box). If this is nonzero, a RST 7 instruction calls the hook table with index 3Ch = 60d. Normally this is an illegal function, but it can be redefined if you want. If location F638h = 63,032d is zero, the

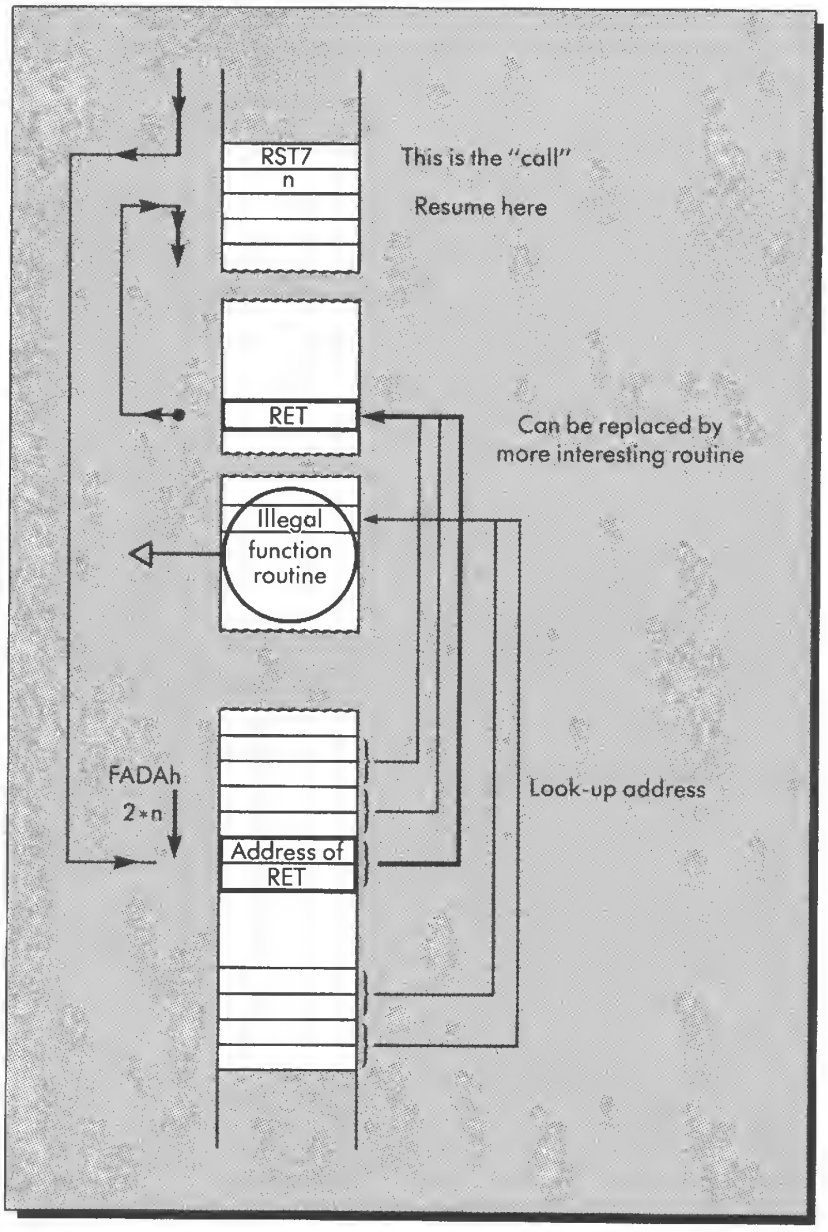


Figure 4-11. How the hook table works

level 4 character-plotting routine is called. Thus location F638h = 63,032d can be used as a device flag for substituting a new device for console output.

Routine: Character Plotting — Level 3

Purpose: To print a character on the LCD screen

Entry Point: 431Fh = 17,183d

Input: Upon entry, the ASCII code of the character to be printed is in the A register.

Output: To the screen

BASIC Example:

```
CALL 17183,A
```

where A is the ASCII code for the character to be printed.

Special Comments: None

Level 4

The level 4 character-plotting routine is located at 4335h = 17,205d (see box). It turns off the clock-cursor-keyboard background task, calls the level 5 character-plotting routine, adjusts the cursor if it is enabled, and then turns the clock-cursor-keyboard background task back on. Location F63Fh = 63,039d stores the cursor enable flag.

Routine: Character Plotting — Level 4

Purpose: To print a character on the LCD screen

Entry Point: 4335h = 17,205d

Input: Upon entry, the ASCII code of the character to be printed is in the C register.

Output: To the screen

BASIC Example: Not directly applicable

Special Comments: This routine cannot be CALLed from BASIC because the ASCII code must be in the C register.

Level 5

The level 5 character-plotting routine at 434Ch = 17,228d handles control characters and escape sequences (see box). If the character is not a control character and is not part of an escape sequence, it calls the level 6 character-plotting routine and advances the cursor.

Routine: Character Plotting — Level 5

Purpose: To print a character on the LCD screen

Entry Point: 434Ch = 17,228d

Input: Upon entry, the ASCII code for the character to be printed is in the C register.

Output: To the screen

BASIC Example: Not directly applicable

Special Comments: None

Control Characters and Escape Sequences

The control characters for the Model 100 include bell (ASCII 7), backspace (ASCII 8), tab (ASCII 9), linefeed (ASCII 10), formfeed (ASCII 12), carriage return (ASCII 13), and delete (ASCII 7Fh = 127d). The level 5 character routine looks for some of these control characters directly and checks for others in a special table at 438Ah = 17,290d (see Appendix K for a complete list).

Escape (ASCII 27) is the control character to initiate an escape sequence. An escape sequence consists of the escape character followed by a sequence of ASCII codes which are to have a special meaning. The Model 100 has a nice set of escape sequences. For example, escape followed by the ASCII code for E is used to indicate that the screen should be erased, and escape followed by the ASCII code for Y and then by two more bytes moves the cursor to a position specified by those two bytes.

The Model 100 looks for escape sequences in a table at 43B8h = 17,336d (see Appendix L). When it detects an escape, it sets a flag (stored at location F646h = 63,046d) so that the next character will be looked up in that table. Some escape sequences have only two characters (for example: escape, E for clearing the screen), and some have more (for example: escape, Y, byte, byte for direct cursor addressing).

Some escape sequences duplicate the actions of the control characters. For example, formfeed (ASCII 12) and the escape "E" sequence both call the routine at 4548h = 17,736d, which clears the screen and puts the cursor into the home position.

A number of escape sequences and control characters can be generated by calling routines (see Appendix M). These are not the routines that actually do the work, such as clearing the screen; instead, they use the RST 4 instruction to send the appropriate control codes and escape sequences. For example, formfeed is generated by calling 4231h = 16,945d, and escape "Q", the escape sequence to turn off the cursor, is generated by calling 424Eh = 16,974d. These routines can be called directly from BASIC programs to perform the indicated functions.

Level 6

The level 6 character-plotting routine is located at 4560h = 17,760d (see box). It puts the ASCII codes for the characters into a special area of memory called the LCD RAM. It also calls the level 7 character-plotting routine to actually plot the character on the screen. The LCD RAM is located from FE00h = 65,024d to FF3Fh = 65,343d. It contains a byte for each character position on the LCD screen. The level 6 character- plotting routine computes the proper address in this RAM and moves the character there so that the current state of the text display on the LCD is always immediately available in this area of main memory.

Routine: Character Plotting — Level 6

Purpose: To print a character on the LCD screen

Entry Point: 4560h = 17,760d

Input: Upon entry, the C register has the ASCII code of the character, and the HL register pair contains the cursor position of the character. The H register contains the column position (1-40), and the L register contains the row position (1-8).

Output: To the screen

BASIC Example: Not directly applicable

Special Comments: None

The LCD RAM is used when the display is scrolled. Scrolling a screen involves rapidly moving characters from one part of the screen to another. When the screen is scrolled on the Model 100, bytes are read not from the LCD drivers but from the LCD RAM area. The scrolling routine is located at 44D2h = 17,618d (see box). It reads the LCD RAM (via a routine at 4512h = 17,682d — see box) to get the characters and then calls the level 6 character-plotting routine (at 4566h = 17,766d instead of 4560h = 17,760d) to place the scrolled characters back on the screen and into the LCD RAM. At the end of the scrolling routine the next line of the display is erased.

Routine: Scroll

Purpose: To scroll part of the LCD screen

Entry Point: 44D2h = 17,618d

Input: Upon entry, the A register contains the scroll count (1-7), and the L register contains the line number (1-7) of the first line to be scrolled. The H register can contain any value. The number of lines scrolled is one more than the scroll count. The sum of the contents of A and L should not exceed 7.

Output: To the screen and the screen RAM (starting at FE00h = 65,024d).

BASIC Example:

```
CALL 17618,N,L
```

where N is the scroll count and L is the first line to be scrolled.

Special Comments: The scrolling always affects one more line than the scroll count. For example, if you scroll one line, starting with line L, then the contents of line L+1 will be moved to line L and line L+1 will be erased.

Routine: Get Character from LCD RAM

Purpose: To get a character from the LCD display

Entry Point: 4512h = 17,682d

Input: Upon entry, the HL register pair contains the cursor position. H contains the column (1-40), and L contains the row (1-8).

Output: When the routine returns, the ASCII code for the character is in the C register.

BASIC Example: Not directly applicable

Special Comments: None

We have included a BASIC program to illustrate how to use the scrolling routine. This program displays a simple message on each line of the screen and then scrolls lines 5 and 6. To stop the program, hit BREAK.

```
100 ' SCROLLING EXAMPLE
110 '
120 ' LABEL THE DISPLAY
130   CLS
140   FOR I = 1 TO 8
150    PRINT "LINE";I;
160    IF I<8 THEN PRINT
170   NEXT I
180 '
190 ' SCROLL LOOP
200   PRINT CH$(27);"Y% SCROLL";J;
210   J=J+1
220   CALL 17618,1,5
230   GOTO 190
```

Looking at the program in detail, we see that it consists of two loops. The first loop (lines 120-170) displays the label "LINE i" on each line, where i is the number of the line. Notice that all but the last line are terminated with a PRINT statement (line 160). The last line is handled differently so that it won't be automatically scrolled up and ruin the display.

The second loop scrolls lines 5 and 6 of the display. On line 200, an escape sequence is used to place the cursor on line 6 of the display, then place the message "SCROLL j" there, where j is the value of a variable J. In line 210, J is incremented. In line 220, the scrolling routine is called, with

the A register set to 1 and the HL register pair set to 5. This forces the scrolling to begin on line 5 with a scroll count of 1. Thus two lines, lines 5 and 6, are affected by the scroll. The program then loops around for the next time through the scroll loop.

Level 7

The level 7 character-plotting routine is located at 73EEh = 29,678d. Notice that this address is quite different from the addresses for the other levels of LCD routines. The 7000h area of memory (above 28,672d), where this routine is located, contains routines that are much more "primitive" and device oriented than routines located in other areas.

Routine: Character Plotting — Level 7

Purpose: To print a character on the LCD screen

Entry Point: 73EEh = 29,678d

Input: Upon entry, the C register contains the ASCII code of the character, and the HL register pair contains the character position. H contains the column (1-40), and L contains the row (1-8).

Output: To the screen

BASIC Example: Not directly applicable

Special Comments: None

The level 7 character-plotting routine looks up the bit patterns for the characters in a table starting at 7711h = 30,481d. This table contains five bytes for the dot matrices for characters whose ASCII codes are in the range 20h = 32d through 7Fh = 127d and six bytes for those in the range 80h = 128d through FFh = 255d. The six-byte part of the table starts at 78F1h = 30,961d. The bytes of this table correspond to the columns of the dot matrix for the characters. This is in contrast to the way CRT character generators usually store the character dot matrix, which is row by row.

The level 7 routine turns off the clock-cursor-keyboard background task, stores the stack pointer in FFF8h = 65,528d, and looks up the character in the previously mentioned character table. The routine takes the bytes from this table and stores the dot matrix for the character in a six-byte area starting at FFECh = 65,516d. If the table entry has only five bytes, the sixth position is filled in as blank. The level 7 routine calls upon a byte-plotting routine at 74A2h = 29,858d, which sends the bytes to the LCD drivers and

concludes by turning on the clock-cursor-keyboard background task with the routine at 473Ch = 19.237d.

The byte-plotting routine computes the information to program the LCD driver (see box). Location FFF4h = 65,524d contains the row, and location FFF5h = 65,525d contains the column, of the character position. A table starting at 7551h = 30,033d (see Appendix N) gives information that should be sent out ports B9h = 185d, BAh = 186d, and FEh = 254d for each horizontal character position on the upper and lower halves of the screen. You may recall from our discussion of point plotting that ports B9h = 185d and BAh = 186d are used to select which of the ten LCD drivers should be enabled and that port FEh = 254d is used to select the byte position within the correct LCD driver. The first two bytes of each entry of this table give the enable information that is sent out ports B9h = 185d and BAh = 186d. The third byte of each entry gives the horizontal position within the LCD driver. Bank selection information from the character row position must be combined with it before it is sent out port FEh = 254d.

Routine: Byte Plotting

Purpose: To send six bytes of a character dot matrix to or from LCD drivers

Entry Point: 74A2h = 29,858d

Input: Upon entry, location FFF4h = 65,524d contains the row and location FFF5h = 65,525d contains the column position of the character to be plotted on the screen. The HL register pair points to the area of memory where the bytes of the dot matrix are stored. This is normally a temporary storage buffer located at FFECh = 65,516d. The D register contains the read/write command. If the bytes are to be sent to the LCD driver, D must contain a 1; otherwise, the bytes are to be read from the LCD drivers.

Output: The character is plotted on the screen. Location FFF6h is affected. It contains a pointer to a table used for selecting the correct LCD drivers.

BASIC Example: Not directly applicable

Special Comments: None

The byte-plotting routine for characters shares a good deal of machine code with the byte-loading routines for plotting points. In particular, it shares the section of code that actually sends the bytes to the LCD drivers.

When it is used to plot characters, the byte-plotting routine sends six bytes at a time to the LCD drivers. These bytes are taken from location FFECh = 65,516d, where they were put by the level 7 character-plotting routine, and sent out port FFh = 255d. The LCD driver accepts them serially, incrementing the horizontal byte position each time. Sometimes, however, the character cell overlaps areas of the screen controlled by two different LCD drivers. The routine is very cleverly designed to send the first few bytes to one driver and the last few bytes to the next driver.

Cursor Blinking

Cursor blinking is controlled as one part of the clock-cursor-keyboard background task. The cursor code starts at 7391h = 29,585d. The actual blink routine starts at 73A9h = 29,609d (see box).

Routine: Cursor Blink

Purpose: To blink the cursor

Entry Point: 73A9h = 29,609d

Input: The temporary dot matrix character buffer at FFECh = 65,516d must contain the dot matrix of the character that is to be blinked. Location FFF3h = 65,523d contains a counter to time the blinking.

Output: To the screen

BASIC Example:

```
CALL 29609
```

Special Comments: The background task must be turned off for this BASIC example to work. Use CALL 30300 to turn off the background task.

First, the cursor routine calls the routine at 765Ch = 30,300d. This turns off interrupt 7.5, which initiates the entire clock-cursor-keyboard task. In this case, however, the routine is called to "rearm" the interrupt. This particular interrupt on the 8085 CPU must be rearmed after each use. Next, the cursor routine checks a counter at FFF3h = 65,523d. The interrupt itself happens every 4 milliseconds, but the counter is set to count down from 125, giving a cursor change every 500 milliseconds. The byte-

plotting routine is used to read six bytes from the LCD drivers at the current cursor position on the screen. These bytes are stored starting at location FFECh = 65,516d. They are then reversed and sent back via the byte-plotting routine. A code of 0 in the D register upon entry to the byte-plotting routine means read from the LCD drivers, and a code of 1 in the D register means write to the LCD drivers.

The cursor routine ends in a return that sends it on to the next part of the background task.

Here's a program that blinks the cursor directly. It quickly blinks the cursor 100 times and then exits. You can change its timing to make the cursor behave in any way you want.

```
100 ' BLINK THE CURSOR
110 '
120   CALL 30300
130   FOR I = 1 TO 100
140    CALL 29609
150    FOR J = 1 TO 20:NEXT J
160   NEXT I
```

On line 120 of this program, the background task is disabled because it too blinks the cursor. The rest of the program consists of a FOR loop (lines 130-160) in which the blink routine is called (line 140) and a short delay is made between blinks (line 150). The delay is made with a FOR loop that counts to twenty. To make the cursor blink faster or slower, change the count in this FOR loop.

Summary

In this chapter we have seen how the LCD works and how it is programmed. In particular, we have seen how it plots points, lines, boxes (filled and unfilled), and characters. We have seen that character plotting is a multilevel process in which the top levels are machine-independent and allow for other devices to be attached for console output, while the bottom levels are very much dependent upon the peculiarities of the LCD display. We have also taken a look at the code in the background task that makes the cursor blink.

5

Hidden Powers of the Real-Time Clock

Concepts

- How the real-time clock works
- What happens when you read the time or date
- What happens when you set the time or date
- BASIC time interrupt commands
- The clock-cursor-keyboard background task

The real-time clock provides the Model 100 with a way to tell both the regular "wall clock" time and the calendar date. This allows you to write programs that do things at prescribed times, making your Model 100 into a valuable assistant for reminding you about appointments and other things you have to do. The timing feature can also be used to control equipment, turning it on and off according to whatever rules you program.

The clock also generates a timing pulse that the Model 100 uses to keep a *background task* going. This background task performs a number of vital functions such as blinking the cursor, updating the system time for the ON TIME$ interrupt, maintaining the automatic power shutoff, and scanning the keyboard.

In this chapter we'll explore the secrets of the real-time clock in the Model 100. We'll start with the hardware and then see how it can be used to set and read the computer's time and date, how it helps control the ON TIME$ interrupt, and how it is used in connection with the clock-cursor-keyboard background task.

How the Real-Time Clock Works

The real-time clock in the Model 100 is housed in a chip called μ PD 1990 AC. The time and date can be written to and read from this clock by sending bits through various CPU ports. The clock chip also outputs a timing pulse that triggers an interrupt to drive the clock-cursor-keyboard background task. A crystal keeps this clock ticking by feeding it electrical pulses at a constant predetermined rate. For the Model 100, the crystal for the clock oscillates at 32,768 cycles per second.

The clock chip contains a series of counters that count seconds, minutes, hours, days of the week, days of the month, and months (see Figure 5-1).

Every 32,768 "ticks" of the crystal causes the seconds counter to increment by one; every time the seconds counter reaches 60, it is zeroed and the minutes counter is incremented; and so on through the hours, days, and months. The chip itself does not have a counter for years.

Most of these counters have a regular cycle. However, the job of the days-of-the-month counter is harder because different months have different numbers of days. The chip is specially preprogrammed to handle this.

It is important to realize that the counting goes on independently of the CPU. Thus, it is not influenced by whatever kind of programs are running on the machine. As a result it can keep accurate time.

A forty-bit *shift register* is connected to the time and date registers to assist with transferring information to and from the time and date counters. In general, a shift register is a row of bit "cells" with provision for *serial transfer* operations, in which binary information is shifted left and/or right along the row. There are also *parallel transfer* operations, in which all the bit cells can be loaded or unloaded (read) at once.

Let's look at the way the bits in the shift register are assigned (see Figure 5-2). The forty bits in this register form ten sets, each containing four bits. The first set holds the units digit of the seconds, the second set holds the tens digit of the seconds, the third set holds the units digit of the minutes, the fourth set holds the tens digit of the minutes, the fifth set holds the units

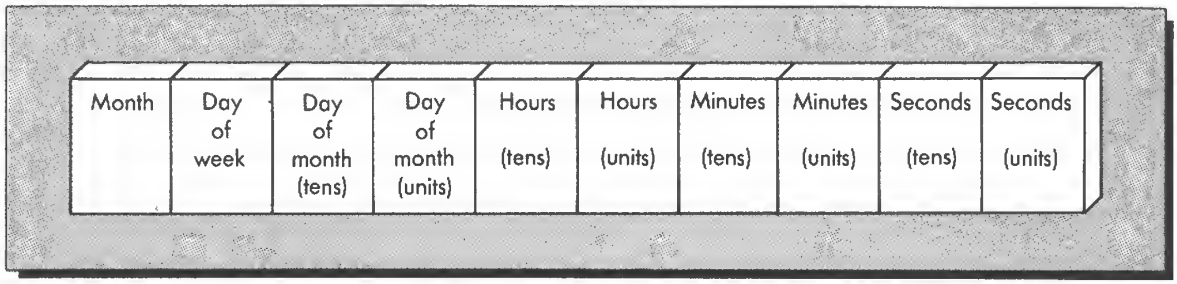


Figure 5-1. Time and date counters

digit of the hour, the sixth set holds the tens digit of the hour, the seventh set holds the units digit of day of the month, the eighth set holds the tens digit of the day of the month, the ninth set holds the day of the week, and the tenth set holds the month. Each set of four bits is a binary coded decimal digit except for the last, which is a hexadecimal encoding of the month.

When the time and date are read from the clock chip, all the bits are transferred at once (in parallel) from the time and date counters to this forty-bit shift register. Then the bits are shifted out of the shift register one by one (serially). Thus the time is "sampled" at a single instant and then moved through the computer bit by bit. Conversely, when the time is set on the clock chip, the individual bits of the time and date are first shifted into the forty-bit shift register one by one (serial transfer), and then the contents of the shift register are transferred all at once to the various time and date registers (parallel transfer) (see Figure 5-3). Other clock chips use other methods for transferring information in and out of the chip. For example, some clock chips transfer the time as a series of bytes rather than a series of bits.

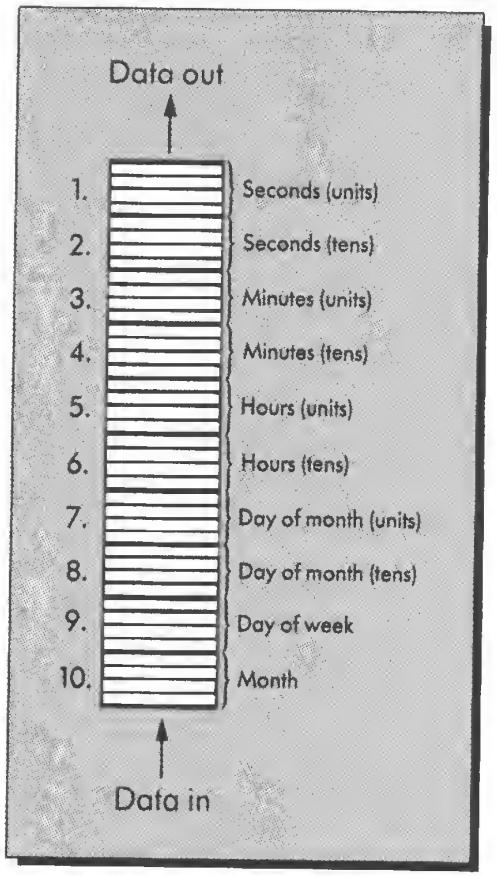


Figure 5-2. The forty-bit shift register in the clock

The clock chip is connected to a number of bits in three different ports of the Model 100 (see Figure 5-4). Three of these, C2, C1, and C0, are command bits and are connected to bits 2, 1, and 0 of port B9h = 185d. These bits form a three-bit binary number that specifies the mode of operation for the chip. When bit C2 is zero, the commands control reading and writing of the real-time clock, and when bit C2 is one, the commands control the timing pulse (see Table 5-1). The clock commands 0 (no operation), 1 (serial transfer mode), 2 (parallel transfer to set the time and date), and 3 (parallel transfer to read the time and date) must be used in combination to set and read the time and date.

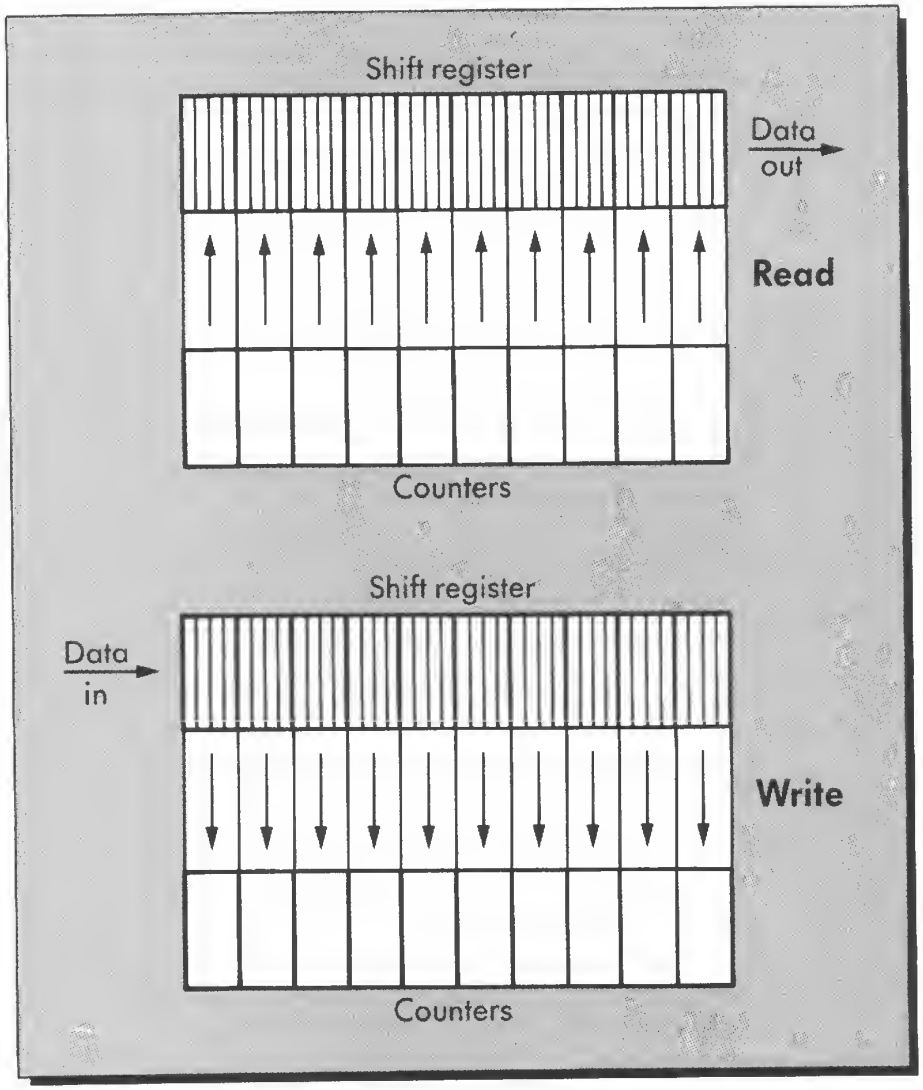


Figure 5-3. Read and write operations of the clock chip

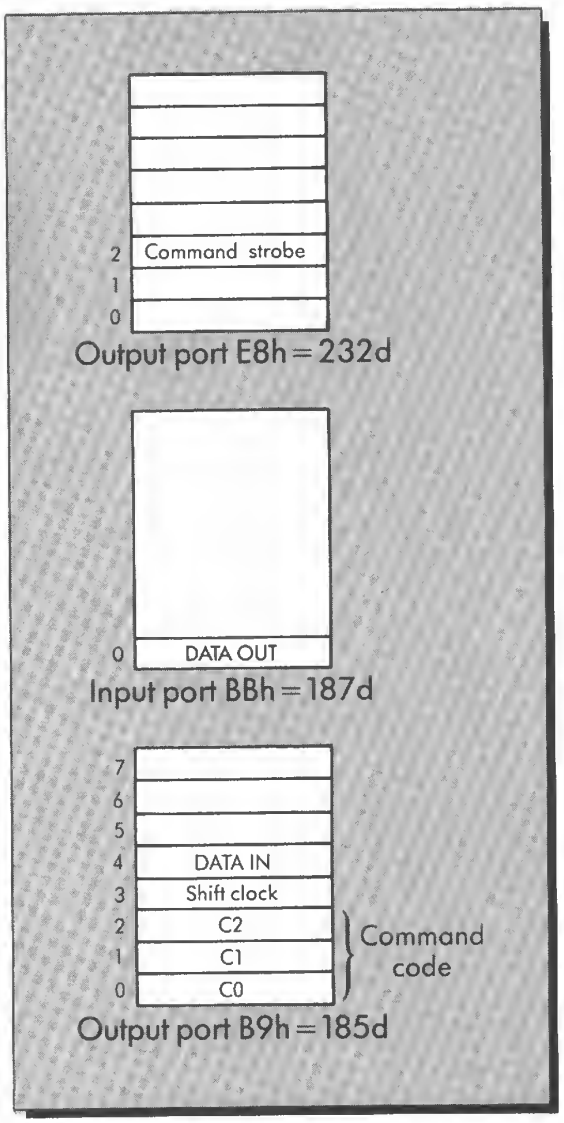


Figure 5-4. Ports for the clock

Command Code	Function
0	No operation
1	Serial transfer
2	Write
3	Read
4–7	Set TP timing

Table 5-1. Clock commands

A command strobe bit is connected to bit 2 of port E8h = 232d. The purpose of the command strobe is to provide "handshaking" for loading commands into the chip. Each pulse on the command strobe causes a new command to be read into the command bits C2, C1, and C0 in port B9h = 185d.

You can send commands to the clock by sending a byte out port B9h = 185d in which bits 2, 1, 0 form the desired command code; and then strobing the command into the chip by sending a byte to port E8h = 232d in which bit 2 is a one, and then a byte to port E8h in which bit 2 is a zero (see Figure 5-5). Later we will examine a routine that does this.

The forty-bit shift register has a data input bit, a data output bit, and data clock input bit. The data input bit is connected to bit 4 of port B9h = 185d, the data output bit is connected to bit 0 of port BBh = 187d, and the data clock bit is connected to bit 3 of port B9h = 185d. The purpose

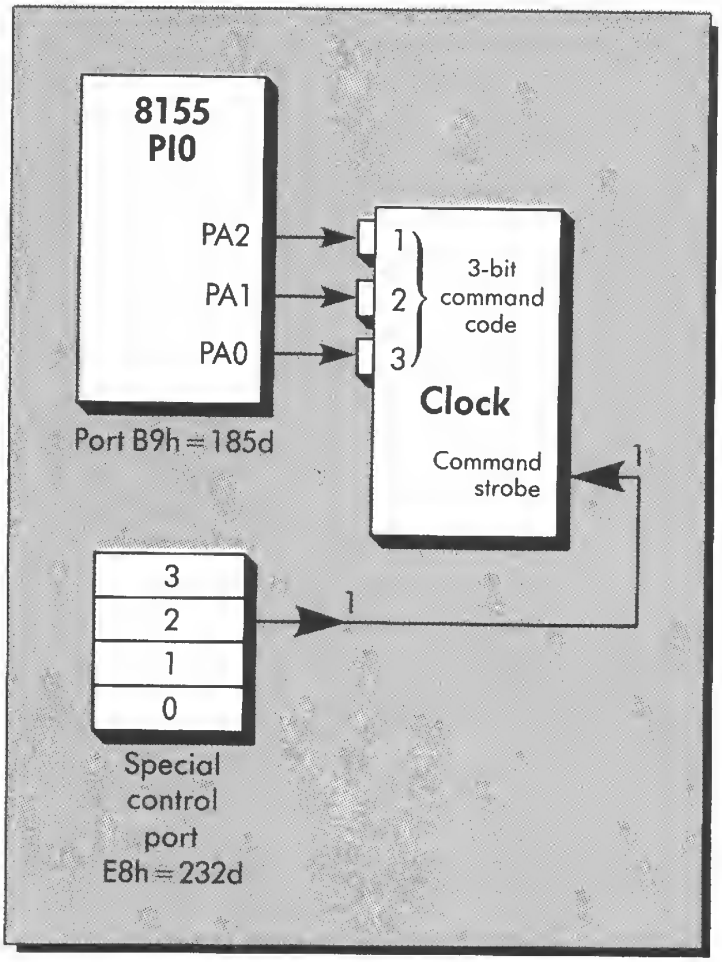


Figure 5-5. Sending a command

of the data clock is to control the serial shift operation. When the serial transfer mode has been selected, each pulse on the data clock bit forces the forty-bit shift register to shift by one place, shifting in one bit from the data input and shifting out one bit into the data output. We will describe this process further in the next two sections.

The ROM Routines

The ROM routines for the clock fall into four classes: primitive command routines, routines to set the clock, routines to read the clock, and the code for the background task.

The Primitive Command Routine

At the lowest level is a routine to send commands to the clock chip. This routine is located at 7383h = 29,571d of the Model 100's ROM (see box). In the next sections we'll see how this routine is used to read and set the time and date on the Model 100. You can also use this routine to control the clock directly from BASIC or machine language.

Routine: Clock Command

Purpose: To send a command to the clock

Entry Point: 7383h = 29,571d

Input: Upon entry, the A register must contain the code for the clock command. A value of 0 means no operation, a value of 1 means put the clock chip into register shift mode, a value of 2 means transfer the time and date from the shift register into the clock counters, and a value of 3 means read the time and date from the clock counters to the shift register.

Output: The clock is programmed accordingly.

BASIC Example:

```
CALL 29571,A
```

where A is the clock command.

Special Comments: This is the most primitive level of programming the clock.

What Happens When You Read the Time or Date

Let's begin with the routines to read the clock, starting at the highest levels (BASIC commands) and working our way down to the actual clock chip routines. The BASIC variables TIME$ and DATE$ are used to read and set the time and date. To read the time or date from BASIC (while running a program or in command mode), you must cause the corresponding variable to be "evaluated". This means that the variable must be part of an expression that might appear on the right side of an equals sign, in an IF clause, or in a PRINT statement.

You can use the information presented in this section to write your own BASIC or machine-language programs to read the clock chip.

The Time

Like the LCD routines to print a character discussed in the last chapter, the time and date routines are composed of a number of different levels. The highest level is designed to execute BASIC commands or functions, while the lowest level directly controls the physical device — in this case, the clock chip. The levels are numbered from top to bottom: the highest level is called level one, the next lower level is called level two, and so on.

The ROM routine to "evaluate" the BASIC variable TIME$ is located at 1904h = 6404d (see box). This is the level 1 time-reading routine. It calls a level 2 time-reading routine and then stores the resulting time in the proper location.

Routine: Read Time — BASIC Command (Level 1)

Purpose: To read the time from the clock chip

Entry Point: 1904h = 6404d

Input: None

Output: When the routine returns, the time is stored as a string whose address is stored at location FB8Ah = 64,394d. Some other locations used in handling string variables are also affected.

BASIC Example:

```
CALL 6404
```

Special Comments: Every time this call is made, a three-byte string descriptor is placed in memory, starting at FB6Eh = 64366d. This can be done only eight times before BASIC runs out of room and declares a ST (string too complex) error.

Here is a BASIC program that exercises the level 1 time-reading routine. It displays a number of locations that are affected by this routine. It even displays the time string itself.

```
100 ' TEST LEVEL 1 TIME READING
110 '
120   CALL 6404
130   X0=PEEK(64393)
140   X1=PEEK(64394)+256*PEEK(64395)
150   X2=PEEK(64396)+256*PEEK(64397)
160   Y =PEEK(64361)+256*PEEK(64362)
170   Z0=PEEK(Y-3)
180   Z1=PEEK(Y-2)+256*PEEK(Y-1)
190   PRINT USING "##";X0;
200   PRINT USING "######";X1;
210   PRINT USING "######";X2;
220   PRINT USING "######";Y;
230   PRINT USING "##";Z0;
240   PRINT USING "######";Z1;
250   PRINT " ";
260   FOR I = Z1 TO Z1+7
270    PRINT CHR$(PEEK(I));
280   NEXT I
290   PRINT
300   GOTO 120
```

On line 120 of this program, the level 1 time-reading routine is called. On lines 130-180, we PEEK at various values, and on lines 190-290 the values are assembled into a display line on the screen.

Let's examine the various PEEKs. The contents of location FB89h = 64,393d is placed in the variable X0. The value is 8, which is the length of the time string. The contents of locations FB8Ah = 64394d and FB8Bh = 64395d form a 16-bit integer that is placed in the variable X1. This value is the address of the time string. The contents of locations FB8Ch = 64396d and FB8Dh = 64397d form a 16-bit integer that is placed in the variable X2. This value is always one less than X1. The contents of locations FB69h = 64361d and FB6Ah = 64362d form a 16-bit integer that is placed in the variable Y. This value points just beyond the three-byte descriptor for the time string. Location Y-3 contains the length of the string and is placed in Z0. Locations Y-2 and Y-1 form a 16-bit integer that is placed in the variable Z1. This is also the address of the string.

The display line first shows the value of X0, then X1, then X2, then Y, then Z0, then Z1, and finally the contents of the string at Z1 through Z1 + 7.

Returning to the ROM routines, we find that the level 2 routine to read the time is located at 190Fh = 6415d (see box). It calls a level 3 time routine,

which reads the raw time and date data into a 10-byte area of memory starting at F923h = 63,779d (see Figure 5-6). Each set of four bits from the shift register is placed in a different byte of memory in the order it comes out of the shift register. Once the raw data is loaded into memory, the level 2 routine turns the time part of this raw data into a string of ASCII numerals with the hours, minutes, and seconds separated by colons. Then it calls a routine at 1996h = 6550d (see box) to fetch the digits and put them in the string.

Routine: Read Time — Level 2

Purpose: To read the time from the clock chip

Entry Point: 190Fh = 6415d

Input: Upon entry, the HL register pair contains the address of an eight-byte area of memory where the string will be stored.

Output: When the routine returns, the time string is stored in the eight-byte area of memory.

BASIC Example:

```
CALL 6415,0,H
```

where H points to an area of memory where the time string will be stored.

Special Comments: None

Here is a BASIC program that explores the level 2 time-reading routine. It places the time in a string variable that we have under our control. Then it repeatedly calls the time routine and prints out the contents of this string variable.

```
100 ' LEVEL 2 TIME READING
110 '
120   T$ = "        "
130   T  = VARPTR(T$)
140   H  = PEEK(T+1)+256*PEEK(T+2)
150   CALL 6415,0,H
160   PRINT T$
170   GOTO 150
```

Looking more closely at this program, we see that on line 120, some blank space is reserved in the string variable T$. In line 130 we find the address of the three-byte string descriptor for T$. In line 140, the address where the string is actually located is given. In line 150, the level 2 time-reading routine is called to dump the time into T$, using the variable H to pass the address. In line 160, T$ is printed. Line 170 loops around to line 150, where the CALL to the time routine is.

The level 3 time-reading routine is located at 19A0h = 6560d (see box). It points to the raw time and date data by placing the address F923h = 63,779d in the HL register, disables interrupts, calls a level 4 routine to get the raw time and date data, and then enables the interrupts before returning.

Routine: Read Time and Date — Level 3

Purpose: To read the time and date from the clock chip

Entry Point: 19A0h = 6560d

Input: None

Output: When the routine returns, the raw time data is in a ten-byte area of memory starting at F923h = 63,779d.

BASIC Example:

```
CALL 6560
```

Special Comments: None

Here is a BASIC program that illustrates the level 3 time and date reading routine. It loops around and around, calling the level 3 time and date reading routine and then printing out the raw time and date data.

```
100 ' LEVEL 3 TIME AND DATE READING
110 '
120   CALL 6560
130   FOR I = 63779 TO 63788
140    PRINT PEEK(I);
150   NEXT I
160   PRINT CHR$(13);
170   GOTO 120
```

Looking more closely, on line 120, the level 3 time and date reading routine is called. On lines 130-150 the raw time and date data are displayed. The first digit is the units digit of the seconds, the second digit is the tens

digit of the seconds, the third digit is the units digit of the minutes, the fourth digit is the tens digit of the minutes, the fifth digit is the units digit of the hours, the sixth digit is the tens digit of the hours, the seventh digit is the units digit of the day of the month, the eighth digit is the tens digit of the day of the month, the ninth digit is the the day of the week, and the tenth digit is the month in hex.

On line 160, a carriage return (ASCII 13) is printed, returning the cursor to the beginning of the display line. This keeps the display on one display line rather than producing a long sequence of lines of output that scroll by. On line 170 the program loops back to the CALL command.

The level 4 time-reading routine is located at 7329h=29,481d (see box). This is in the higher area of the ROM, where other low-level, machine-dependent routines are also located.

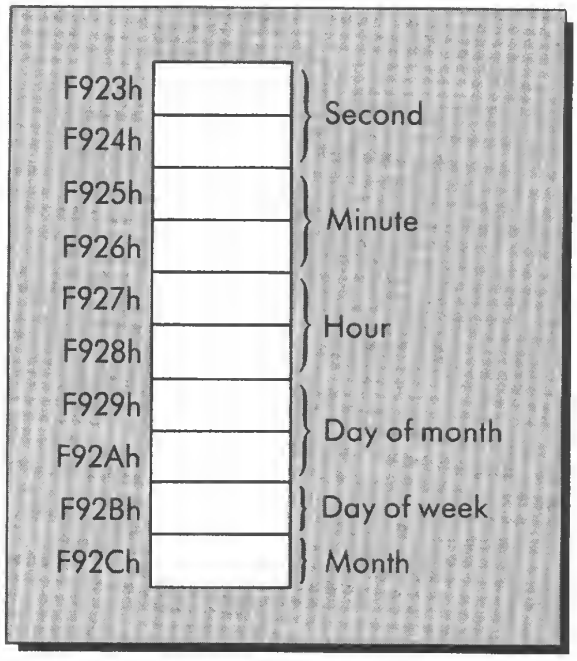


Figure 5-6. Raw time and date data

Routine: Read Time and Date — Level 4

Purpose: To read the time and date from the clock chip

Entry Point: 7329h = 29,481d

Input: Upon entry, the HL register pair points to a ten-byte area of memory.

Output: When the routine returns, the raw time and date data are in the ten-byte area of memory.

BASIC Example:

```
CALL 29481,0,H
```

Special Comments: None

We have a BASIC program that illustrates this level as well. It sets up a string to store the raw time and date data, calls the level 4 routine to dump the raw time and date in this string, and then prints out the raw data.

```
100 ' LEVEL 4 TIME AND DATE READING
110 '
120   T$ = "            "
130   T  = VARPTR(T$)
140   H  = PEEK(T+1)+256*PEEK(T+2)
150   CALL 29481,0,H
160   FOR I = H to H+9
170    PRINT PEEK(I);
180   NEXT I
190   PRINT
200   GOTO 150
```

This program is a combination of the previous two programs. On lines 120-140, it sets up the string T$ for storage. On line 150, it calls the level 4 time and date reading routine. On lines 160-190, it prints out the raw data. On line 200, it loops back for more.

First the level 4 routine strobes the read command into the chip by putting the value 3 into the A register and calling the clock command routine at 7383h = 29,571d (discussed in the previous section). In response to this command, the chip does the parallel transfer from the time and date counters to the forty-bit shift register.

Next the level 4 routine strobes the serial transfer command into the chip by calling the clock command routine again, this time with 1 in the A register. Now the system is ready to read the time and date bit by bit.

Each bit of the time and date is obtained by capturing bit 0 of port BBh = 187d. The IN BBh command gets the byte into the A register, the A register is shifted one place to the right to put the bit into the carry flag, and the contents of the carry is shifted into the D register with three more instructions. After each bit is read, the shift register is shifted by strobing the data clock (bit position 3 of register B9h = 185d). This strobe is accomplished by moving a byte into register B9h = 185d that has a zero in bit position 3; then a byte that has a one in this bit position is moved into this same port. The data bits from the shift register are collected in groups of four, one group for each of the ten digits of the raw time and date data. After all the bits are read, a "no operation" command is strobed into the chip by calling the clock command routine with zero in the A register. This stops the serial transfer.

The Date

The routine to "evaluate" the BASIC string variable DATE$ (see box) works in much the same way as the TIME$ routine. It is located at 1924h = 6436d. It calls a level 2 date routine and stores the resulting string in the appropriate place.

Routine: Read Date — BASIC Command (Level 1)

Purpose: To read the date from the clock chip

Entry Point: 1924h = 6436d

Input: None

Output: When the routine returns, the date is stored as a string whose address is placed at location FB8Ah = 64,394d. Some other locations used in handling string variables are also affected.

BASIC Example:

```
CALL 6436
```

Special Comments: As with the TIME$ routine, every time this call is made, a three-byte string descriptor is placed in memory, starting at FB6Eh = 64366d. This can be done only eight times before BASIC runs out of room and declares a ST (string too complex) error.

Here is a BASIC program that exercises the level 1 date-reading routine. It is almost identical to the level 1 time-reading program given previously: the only difference is the address of the routine.

```
100 ' TEST LEVEL 1 DATE READING
110 '
120   CALL 6436
130   X0=PEEK(64393)
140   X1=PEEK(64394)+256*PEEK(64395)
150   X2=PEEK(64396)+256*PEEK(64397)
160   Y =PEEK(64361)+256*PEEK(64362)
170   Z0=PEEK(Y-3)
180   Z1=PEEK(Y-2)+256*PEEK(Y-1)
190   PRINT USING "##";X0;
200   PRINT USING "######";X1;
210   PRINT USING "######";X2;
220   PRINT USING "######";Y;
230   PRINT USING "##";Z0;
240   PRINT USING "######";Z1;
250   PRINT " ";
260   FOR I = Z1 TO Z1+7
270    PRINT CHR$(PEEK(I));
280   NEXT I
290   PRINT
300   GOTO 120
```

Since this is almost the same as the earlier program, we will not discuss its structure.

The level 2 date routine (see box) at 192Fh = 6447d calls the level 3 routine at 19A0h = 6560d to read the time and date described above. Recall that this routine puts the raw time and date data in a ten-byte area of RAM starting at location F923h = 63,779d. After this, the level 2 date routine uses the raw data to create a string with the month, day, and year in ASCII numerals separated by slashes. The month has to be dealt with in a slightly different way than the other parts of the date because it is expressed in hexadecimal notation rather than decimal.

Routine: Read Date — Level 2

Purpose: To read the date from the clock chip

Entry Point: 192Fh = 6447

Input: Upon entry, the HL register pair contains the address of an eight-byte area of memory where the string will be stored.

Output: When the routine returns, the date string is stored in the eight-byte area of memory.

BASIC Example:

```
CALL 6447,0,H
```

where H points to an area of memory where the date string will be stored.

Special Comments: None

Here is a BASIC program that illustrates the level 2 date-reading routine. It is the same as the one for the level 2 time-reading routine except for line 150, where the level 2 date routine is called instead of the level 2 time routine.

```
100 ' LEVEL 2 DATE READING
110 '
120   T$ = "          "
130   T  = VARPTR(T$)
140   H  = PEEK(T+1)+256*PEEK(T+2)
150   CALL 6447,0,H
160   PRINT T$
170   GOTO 150
```

The Day of the Week

There is also a routine to "evaluate" the BASIC string variable DAY$, which returns the day of the week as a three-character string (see box). The routine is located at 1955h = 6485d. It calls a level 2 day routine and places the resulting string in the DAY$ variable.

Routine: Read Day of Week — BASIC Command (Level 1)

Purpose: To read the day of the week from the clock chip

Entry Point: 1955h = 6485d

Input: None

Output: When the routine returns, the day is stored as a string whose address is placed in location FB8Ah = 64,394d. Some other locations used in handling string variables are also affected.

BASIC Example:

```
CALL 6485
```

Special Comments: Every time this call is made, a three-byte string descriptor is placed in memory, starting at FB6Eh = 64,366d. This can be done only eight times before BASIC runs out of room and declares a ST (string too complex) error.

The level 2 day routine is located at 1962h = 6498d (see box). It calls the level 3 time and date reading routine at 19A0h = 6560d (described previously). It then plucks out the numerical value for the day of the week from the raw time and date data and looks at the corresponding three-byte string in a table starting in the ROM at 1978h = 6520d.

Routine: Read Day of Week — Level 2

Purpose: To read the day of the week from the clock chip

Entry Point: 1962h = 6498d

Input: Upon entry, the HL register pair contains the address of a three-byte area of memory where the string will be stored.

Output: When the routine returns, the day string is stored in the three-byte area of memory.

BASIC Example:

```
CALL 6498,0,H
```

where H points to an area of memory where the day string will be stored.

Special Comments: None

It is easy to modify the level 1 and level 2 BASIC programs given for the time and date so that they can work for the day of the week. We invite you to do that yourself.

What Happens When You Set the Time or Date

To set the time, date, or day of the week you must assign an appropriate string expression to the corresponding variable. This happens when the variable TIME$, DATE$, or DAY$ appears on the left side of an equals sign.

The Time

The BASIC routine to set the time (see box) is the reverse of the routine discussed above to evaluate TIME$. The level 1 routine is located at 19B0h = 6576d. It checks for an equals sign following the TIME$ symbol. Then it calls a routine at 1A42h = 6722d (see box), which evalutes the string expression on the right side of the equals sign, calls the level 3 time and date reading routine to read the raw time and date data from the clock into RAM starting at F923h = 63,779d, and then replaces the time part of the raw data by the appropriate new data. The level 1 routine finishes by calling a level 2 time and date setting routine to place the modified raw data back into the clock.

Routine: Set Time — BASIC Command (Level 1)

Purpose: To set the time on the clock chip

Entry Point: 19B0h = 6576d

Input: Upon entry, the HL register pair points to a command line containing a string expression that evaluates a time string.

Output: When the routine returns, the specified time is set in the clock chip.

BASIC Example:

```
CALL 6576,0,H
```

where H is the address of a valid time string.

Special Comments: None

Here is a BASIC program that shows how to use the level 1 time-setting command. It contains an infinite loop that gets a time string from the user, sends it to the time-setting routine, and then displays the new time using the BASIC TIME$ function to verify that the time has actually been set. When you run it, be sure to type in the time in exactly the same format as the Model 100 prints out the time.

```
100 ' LEVEL 1 TIME SET
110 '
120   INPUT "TIME";T$
130   S$=CHR$(221)+"T$"+CHR$(0)
140   S = VARPTR(S$)
150   H = PEEK(S+1)+256*PEEK(S+2)
160   CALL 6576,0,H
170   PRINT TIME$
180   GOTO 120
```

The main loop of this program extends over lines 120-180. On line 120, the user inputs the time into the variable T$. On line 130, the time is encased in a command line and stored in the variable S$. In lines 140-150, the address of the string is computed and stored in the variable H. On line 160, the time-setting routine is called. On line 170, the program verifies that the time has been set by printing out the TIME$ variable. On line 180 it loops back for another time.

Routine: Get Time String from Command Line

Purpose: To get the time string from the command line

Entry Point: 1A42h = 6722d

Input: Upon entry, the HL register pair points to a command line containing a valid time string.

Output: When the routine returns, the new raw time and date data are stored in a ten-byte area in memory starting at F923h.

BASIC Example: Not directly applicable

Special Comments: This routine uses the stack and therefore cannot be CALLed directly from BASIC.

The level 2 time and date setting routine is located at 732Ah = 29,482d, in the high part of the ROM, and thus is considered a low-level, machine-dependent routine. It shares much of its code with the level 4 time and date reading routine at 7329h = 29,481d (described previously). However, it first strobes the serial transfer command into the chip (by calling the command routine at 7383h = 29,571d with 1 in the A register). Then it transfers each bit of the time and date in through bit position 2 of port B9h = 185d, strobing the data clock (bit 3 of port B9h = 185d) each time. It finishes by strobing the write command and then the no operation command into the chip.

The Date

The BASIC routine to set the date (see box) works in a similar way. It is located at 19BDh = 6589d and calls many of the same routines.

Routine: Set Date — BASIC Command (Level 1)

Purpose: To set the date on the clock chip

Entry Point: 19BDh = 6589d

Input: Upon entry, the HL register pair points to a command line containing a string expression that evaluates to a date string.

Output: When the routine returns, the specified date is set in the clock chip.

BASIC Example:

```
CALL 6589,0,H
```

where H is the address of a valid date string.

Special Comments: None

Here is a BASIC program that shows how to use the level 1 date-setting command. When you run it, be sure to type in the date in exactly the same format as the Model 100 prints out the date. Because it is almost the same as the level 1 time-setting program described earlier, we will not discuss it in detail.

```
100 ' LEVEL 1 DATE SET
110 '
120   INPUT "DATE";T$
130   S$=CHR$(221)+"T$"+CHR$(0)
140   S = VARPTR(S$)
150   H = PEEK(S+1)+256*PEEK(S+2)
160   CALL 6589,0,H
170   PRINT DATE$
180   GOTO 120
```

The Day of the Week

There is also a BASIC routine to set the day of the week (see box), located at 19F1h = 6641d. You might want to modify the program above to work for this DAY$ function.

Routine: Set Day of Week — BASIC Command (Level 1)

Purpose: To set the day of week on the clock chip

Entry Point: 19F1h = 6641d

Input: Upon entry, the HL register pair points to a command line containing a string expression that evaluates to a day string.

Output: When the routine returns, the specified day of the week is set in the clock chip.

BASIC Example:

```
CALL 6641,0,H
```

where H is the address of a valid day string.

Special Comments: None

BASIC Time Interrupt Commands

BASIC has certain commands that allow you to make your portable computer into a fancy alarm clock or a controller for lab equipment. These commands are ON TIME$...GOSUB, TIME$ ON, TIME$ OFF, and TIME$ STOP. They control a BASIC interrupt that is triggered by the time of day.

The ON TIME$...GOSUB command allows you to specify a time and a BASIC subroutine that you want called at that specified time. The routine to handle this command starts at location 1B0Fh = 6927d (see box). It calls a subroutine at 1AFCh = 6908d (see box) that sets the time for the interrupt. This subroutine calls a routine at 1A42h = 6722d (described previously) to get the time from your BASIC command line and transform it into raw form. Then it calls a block-move routine at 3469h = 13,417d (see box) to transfer it into a six-byte area of RAM starting at F93Dh = 63,805d. In the next section we will see how the clock-cursor-keyboard background task continually examines this six-byte area, looking for a match with the current time.

Routine: ON TIME$...GOSUB — BASIC Command

Purpose: To set the ON TIME$ interrupt

Entry Point: 1B0Fh = 6927d

Input: Upon entry, the HL register pair contains the address of the end of a command line for the ON TIME$ command.

Output: When the routine returns, the location of the ON TIME$ subroutine and the time that it should be executed are loaded into BASIC.

BASIC Example:

```
CALL 6927,0,H
```

where H is the address of the end of the ON TIME$ command.

Special Comments: None

Routine: Block Transfer

Purpose: To transfer bytes from one location to another location in memory

Entry Point: 3469h = 13,417d

Input: Upon entry, the B register contains the number of bytes to be tranferred, the DE register pair points to the source location, and the HL register pair points to the destination location.

Output: When the routine returns, the bytes have been transferred.

BASIC Example: Not applicable

Special Comments: This routine is used many times throughout the Model 100's ROM.

The ON TIME$...GOSUB routine finishes by getting the location of the BASIC line specified after the GOSUB. It puts this location into a two-byte RAM location starting at F948h = 63,816d. This is part of a three-byte area of RAM starting at F947h = 63,815d, which stores information about the time interrupt.

The routines to handle the TIME$ ON, TIME$ OFF, and TIME$ STOP commands all start at 1AA5h = 6821d. Here the HL register is set to point to the first byte of the three-byte area of RAM starting at F947h = 63,815d, which stores information about the time interrupt. These routines call a routine at 1AEAh = 6890d, which in turn branches to individual routines to handle ON, OFF, or STOP (see boxes). These routines are also used to control other interrupts such as the ON KEY$ interrupts, which will be discussed in the next chapter.

Routine: Interrupt ON — BASIC Command

Purpose: To enable the interrupt

Entry Point: 3FA0h = 16,288d

Input: Upon entry, the HL register pair contains the address of the three-byte interrupt table.

Output: When the routine returns, the interrupt is enabled.

BASIC Example:

```
CALL 16288,0,H
```

where H is the address of the three-byte interrupt table.

Special Comments: None

Routine: Interrupt OFF — BASIC Command

Purpose: To disable the interrupt

Entry Point: 3FB2h = 16,306d

Input: Upon entry, the HL register pair contains the address of the three-byte interrupt table.

Output: When the routine returns, the interrupt is disabled.

BASIC Example:

```
CALL 16306,0,H
```

where H is the address of the three-byte interrupt table.

Special Comments: None

Routine: Interrupt STOP — BASIC Command

Purpose: To stop the interrupt

Entry Point: 3FB9h = 16,313d

Input: Upon entry, the HL register pair contains the address of the three-byte interrupt table.

Output: When the routine returns, the interrupt is stopped.

BASIC Example:

```
CALL 16313,0,H
```

where H is the address of the three-byte interrupt table.

Special Comments: None

The first byte of this interrupt storage area (at F947h = 63,815d) is called the "interrupt status byte". Three of its bits are used to manage the ON TIME$ interrupt. Bit 0 tells whether the interrupt is on or off, bit 1 tells whether the interrupt is stopped or not, and bit 2 tells whether the interrupt has occurred. There are six values that this byte normally takes on, depending upon the values of these bits (two more values are possible but never actually occur). Each value represents a *state* for the system with regard to the ON TIME$ interrupt. These states form a "finite state machine" (see Figure 5-7). This term is used by computer scientists to describe a system that has a finite number of states and a set of possible "transitions" between these states. It is a useful concept for understanding everything from the basic electrical circuits that make up a computer to the workings of programs like a sophisticated text editor. Computer scientists use diagrams such as the one in Figure 5-7 to help them visualize the workings of such systems.

Let's look at the various states of this "interrupt machine" in more detail. There are three primary states: 0, 1, and 5. A value of 0 (all bits off) indicates that the interrupt is *off* (cannot cause any action). A value of 1 (just bit 0 on) means that the interrupt is on but has not yet been "triggered" (the clock has not yet reached the time specified for the interrupt). A value of 5 means that the interrupt is on and has actually been triggered. When this happens, we say that the interrupt is *pending*.

For each of these three primary states there is a corresponding "stopped" state. In the stopped states, interrupts are "remembered" but not acted upon. A value of 2 indicates that the interrupt is off and stopped. A value

of 3 indicates that the interrupt is on but stopped. A value of 7 indicates that the interrupt has been triggered but is stopped.

The TIME$ ON, TIME$ OFF, and TIME$ STOP commands, as well as the actual triggering and processing of the ON TIME$ interrupt, cause state *transitions* within this finite state machine. For example, if the system is in state 0 (off), then the TIME$ ON command will cause it to move to state 1 (on). TIME$ OFF, on the other hand, causes the system to move to state 0 (off), no matter what state it was in previously. The arrows in the state diagram show all possible state transitions.

The ON part of the TIME$ routine (described previously) is located at 3FA0h = 16,288d; the OFF part is located at 3FB2h = 16,306d; and the

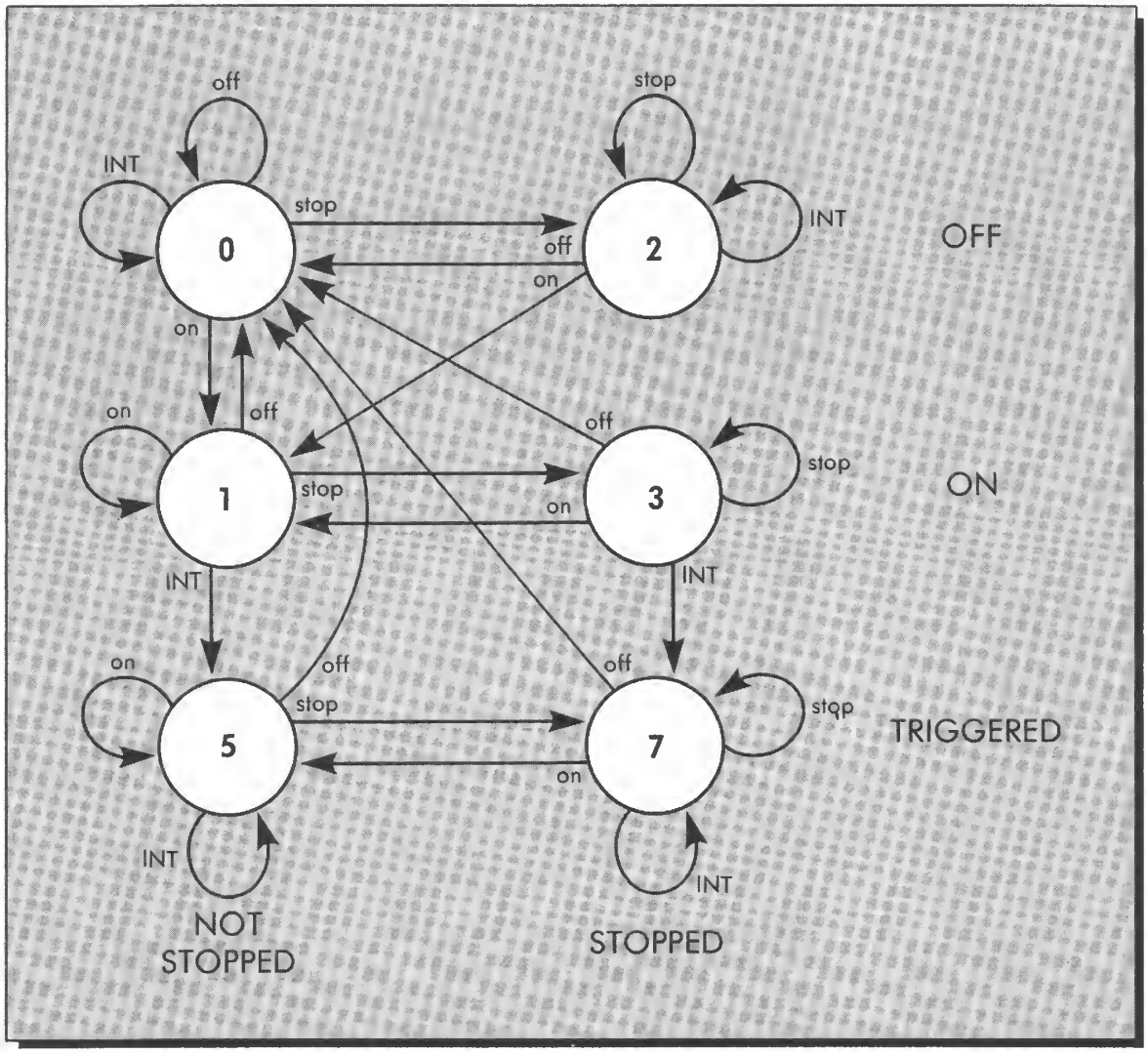


Figure 5-7. Finite state machine for interrupts

STOP part is located at 3FB9h = 16,313d. In addition, there is a routine at 3FD2h = 16,338d (see box), which adjusts this finite state machine each time an interrupt is triggered, and a routine at 3FF1h = 16,369d (see box), which adjusts the state each time the interrupt is processed.

Routine: Trigger Interrupt

Purpose: To trigger the interrupt

Entry Point: 3FD2h = 16,338d

Input: Upon entry, the HL register pair contains the address of the three-byte interrupt table.

Output: When the routine returns, the interrupt is triggered.

BASIC Example:

```
CALL 16338,0,H
```

where H is the address of the three-byte interrupt table.

Special Comments: None

Routine: Clear Interrupt

Purpose: To clear the interrupt

Entry Point: 3FF1h = 16,369d

Input: Upon entry, the HL register pair contains the address of the three-byte interrupt table.

Output: When the routine returns, the interrupt is cleared.

BASIC Example:

```
CALL 16369,0,H
```

where H is the address of the three-byte interrupt table.

Special Comments: None

Besides manipulating the interrupt status byte at F947h = 63,815d, these routines maintain an "interrupt counter byte" at F654h = 63,060d. This byte is incremented each time state 5 (interrupt pending) is entered and decremented each time it is exited (turned off, stopped, or processed). Unlike the status byte at F947h = 63,815d, this byte is shared with other BASIC interrupts such as the ON KEY, ON COM, and ON MDM interrupts. Thus this byte counts the total number of BASIC interrupts of *any* type that are pending in the computer. In Chapters 6 and 7 we will examine other BASIC interrupts.

The Clock-Cursor-Keyboard Background Task

The clock-cursor-keyboard background task helps maintain certain basic functions in the Model 100 computer. These include updating the clock, blinking the cursor, and scanning the keyboard. All these jobs must be performed very often; that is, several times a second to several times a minute. Fortunately they do not take much time to perform; thus, they do not appreciably slow down the main (foreground) tasks that the computer is asked to do.

The clock-cursor-keyboard background task involves the real-time clock in two ways. First, the clock drives the background task, causing it to be performed about 256 times a second; and secondly, the clock is read every 125 times that the background task is called, or about every half second. This is done to support the ON TIME$ interrupt. The clock part of the background task also maintains the automatic power-off feature and the year part of the date.

Generating the Interrupt

The timing pulse from the clock is initialized at 6CEBh = 27,883d in the warm start reset routine (see Chapter 3), which sends a command code 5 to the clock chip (via the clock command routine at 7383h = 29,571d, described previously. This causes the clock chip to pulse at a frequency of 256 times a second, or about once every 4 milliseconds.

The timing pulse is fed into the interrupt 7.5 input pin on the 8085 CPU. This CPU interrupt can be selected using the SIM instruction and enabled and disabled using the EI and DI instructions. In general, the SIM command is used to tell which of three interrupts (5.5, 6.5, and 7.5) will be enabled with the EI (enable interrupt) instruction. If the 7.5 interrupt is enabled, a timing pulse will cause the CPU to stop what it is doing and call the routine at 3Ch = 60d (see Chapter 3). In the Model 100, location

3Ch = 60d has some code that disables interrupts and jumps to 1B32h = 6962d, where the beginning of the clock-cursor-keyboard background task is located. Each time interrupt 7.5 is actuated, it must be rearmed before it can be used again. This is done by the routine at 765Ch = 30,300d (see Chapter 4), which uses the SIM command to turn off and rearm the interrupt. This routine was mentioned in Chapter 4 on the LCD.

The code for the background task comes in three major sections — one for the clock, one for the cursor, and one for the keyboard. We have already discussed the section for the cursor in Chapter 4, and we will discuss the section for the keyboard in Chapter 6. We will now discuss the section for the clock (see box).

The clock-cursor-keyboard background task begins at 1B32h = 6962d and consists of several subsections.

Routine: Clock Section of Background Task

Purpose: To update the system time for the ON TIME$ interrupt, maintain automatic power off, and update the year.

Entry Point: 1B32h = 6962d

Input: None

Output: System time and timing parameters are updated.

BASIC Example: Not applicable

Special Comments: None

The "Very Often" Routine

Before doing anything else, this task calls a routine located in RAM at F5FFh = 62,975d (see box). Normally, a RETurn instruction is located there, so nothing much happens, but if you place your own routine there, it will be called "very often". This "very often" routine can be used to run your own background task in addition to the background task that is already built into the computer. Each execution of the "very often" routine corresponds to one "tick" of the background task.

Routine: Very Often

Purpose: To perform one tick of a background task

Entry Point: F5FFh = 62,975d

Input: None, as it stands

Output: None, as it stands

BASIC Example:

```
CALL 62975
```

Special Comments: None

The bytes at F5FFh = 62,975d are some of the RAM locations that are initialized when the machine is first turned on. Initially, a RETurn followed by two NOP instructions is placed at F5FFh = 62,975d. To install your own "very often" routine replace these three bytes by a JuMP or CALL instruction to a routine that you have placed somewhere else in memory.

A stopwatch program is an example of a way the "very often" routine can be used. Such a program could be written in machine language and CALLed from BASIC. The main stopwatch program would set up a variable in memory for a count and use its own "very often" routine to increment this variable each time it is called. This "very often" would then count the ticks of the system clock (256 times a second). The main stopwatch program would have to monitor some keys on the keyboard (see Chapter 6) to tell when to zero the count, start and stop the count routine, and report the results.

Some Housekeeping

Let's contine the discussion of the Model 100's built-in background task. Right after the "very often" routine is called, the HL, DE, BC, A, and Flags are pushed onto the stack. This is because the background task is run as an interrupt routine and therefore must return with the contents of the CPU registers as they were upon entry. Note that if you install your own "very often" routine, you must make sure that it, too, does not modify any registers.

The next action of the background task is to allow certain interrupts and block others with the SIM command. These are 7.5 (the background task itself), which is blocked; 6.5 (the serial communications line), which is allowed; and 5.5 (the bar code reader), which is blocked. To do this, the A register is loaded with 00001101 binary before the SIM instruction is exe-

cuted. The 1 in bit 3 indicates that the SIM command is being used to select interrupts; bits 2, 1, and 0 select the interrupts, as indicated above.

The Clock Section

The clock part of the clock-cursor-keyboard background task now starts. It begins by decrementing a counter (located at F92Fh = 63,791d) that causes the clock section to be executed about every half second. This works as follows: if the counter does not become zero, the clock section is skipped and the CPU goes onto the next (cursor) section of the background task. However, if the counter reaches zero, the CPU continues into the rest of the clock section, resetting the counter to 125. Since the interrupt happens about every 4 milliseconds, the cycle length of this process is about half a second.

If the rest of the clock section is executed, another counter (at F930h = 63,792d) is decremented. This counter is reset to 12 each time it reaches zero, giving a six-second timing cycle to the section of code immediately following. This particular code takes care of the Model 100's automatic power-off feature.

The Automatic Power Off

To conserve battery power, the Model 100 has an automatic power-off feature. Normally, if the machine is left alone and it is not running a program, it will shut off after about ten minutes. Fortunately, the RAM memory stays on even when the automatic power-off shuts down the rest of the computer, so you will never lose any work because of this feature.

The automatic power-off code starts at 1B4Eh = 6990d. It checks to see if you are running a BASIC program by looking at the current line number (stored at F67Ah = 63,098d). If this variable is not −1, it assumes you are running a program and renews the power-off timer counter (at F631h = 63,025d) by calling a routine at 1BB1h = 7089d (see box). The initial value for the count is stored in F657h = 63,063d.

Routine: Renew Automatic Power-Off Timer

Purpose: To reinitialize the power-off timer

Entry Point: 1BB1h = 7089d

Input: None

Output: When the routine returns, the contents of location F657h = 63,063d (the full count) are moved to location F931h = 63,793d (the counter).

BASIC Example: Not applicable (happens while BASIC is running anyway).

Special Comments: None

The power-off timer counter is decremented each time the automatic power-off code is executed. Since this code is executed about every six seconds or every tenth of a minute, and since the default count value is 100, the default time for power-off is about ten minutes. If the counter is already zero, it is not decremented, and the CPU is sent on past the power-off code. This is designed to handle the case when the user disables the power-off feature with the command POWER CONT.

Location F932h = 63,794d is used to tell the rest of the system when the power should be turned off. If the power-off timer actually decrements from 1 to 0, location F932h = 63,794d is set equal to −1. The main input routine of BASIC (in particular at 1358h = 4952d) examines this location and turns off the power if it becomes nonzero. The actual code to turn off the power is at 143Fh = 5183d.

Detecting the ON TIME$ Interrupt

After the automatic power-off code, the raw time and date data from the clock are read into a ten-byte area of memory starting at F933h = 63,795d. The low-level time and date routine at 7329h = 29,481d, which we discussed earlier, is used.

Next the current time raw data are compared with the time that was specified by the last ON TIME...GOSUB command. If there is no match, it proceeds; otherwise, it calls a routine at 3FD2h = 16,338d (described previously) to further handle the time interrupt. This routine is part of the finite state machine and thus operates upon the time interrupt status byte at F947h = 63,815d and the interrupt counter at F654h = 63,060d, causing them to indicate that the ON TIME$ interrupt has just been triggered.

Updating the Year

The next section takes care of updating the year. It gets the month from where it was just stored by the low-level time and date routine. It puts this in location F655h = 63,061d. If the value is less than or the same as what was previously stored there, the routine proceeds; otherwise, it increments the year stored at F92Dh = 63,789d and F92Eh = 63,790d.

Finally, the clock section checks the optional I/O and jumps to the cursor section.

Summary

In this chapter, we have explored the operation of the real-time clock and the routines that control it. Among these are routines to set and read the time of day, the date, and the day of the week. We have also studied routines to control the ON TIME$ interrupts. Finally, we have studied the background task, which performs a number of functions that have to be performed very often. These include handling the automatic power-off feature, updating the time for the ON TIME$ interrupt, updating the year, and performing other tasks such as cursor blinking (see Chapter 4) and keyboard scanning (see Chapter 6).

6

Hidden Powers of the Keyboard

Concepts

- How a scanning keyboard works
- How to program the keyboard
- Descriptions of ROM routines
- The clock-cursor-keyboard background task
- Keyboard input routines

The keyboard is the main input device for the Model 100. The hardware for the keyboard is quite simple, consisting of an array of switches that correspond to the keys of the keyboard. However, the software that manages the keyboard is quite complex.

In this chapter we'll explore the secrets of the Model 100's keyboard. We will see in general how such a keyboard works; then we will study the specific details of the Model 100's keyboard. We will describe the section of the background task in the Model 100's ROM that runs the keyboard, allowing the computer to continually capture keystrokes as it goes about its other business. We will see how the characters are stored in a buffer as they are typed, and we'll see how other routines pick them up as needed. We will also describe the ON KEY BASIC interrupt routines that allow you to program the Model 100 in an interactive, user-friendly manner.

How a Scanning Keyboard Works

The Model 100 uses a "scanning keyboard" run by the CPU. This kind of keyboard requires that the CPU constantly send pulses into the keyboard

through certain signal lines while "sampling" certain other lines coming out of the keyboard. This "pulsing and sampling" must happen quite frequently because the keyboard hardware cannot "remember" what key or keys you hit once you have released them. On the Model 100 the entire keyboard is scanned every 12 milliseconds. That's about 84 times a second.

From the scanning information the CPU can tell exactly what key or key combinations are being depressed. We will see later how the computer can develop a complete "image" of the keyboard in its memory as it scans the keyboard.

Logically, the keys are arranged in the form of a two-dimensional matrix (see Figure 6-1). Each key operates a switch that connects one input line to one output line. The input lines run through the keyboard forming the columns of this matrix, and the output lines form the rows of the matrix. Each key lies at an intersection and thus is uniquely identified by a choice of one input line and one output line. You should be aware that the *logical* wiring of this matrix does not conform exactly to the *physical* arrangement of the keys.

Let's look at how the sampling process works. We'll start with the job of determining if a single specified key is depressed, and then we will see how to scan for *any* key. To look for a particular key, the first step is to turn "on" the input line for its column, turn "off" the input lines for all the other columns, and then examine the output lines. Under this condition, the output lines tell you the state of the keys of just this one column. If a particular output line is "on", then its key is being depressed; otherwise, it's not.

To find out if any key at all has been depressed, you must check all the columns, one at a time, using the above method. The resulting sets of "on/off" patterns of the columns can be arranged side by side to give a "picture" of the keyboard showing exactly which keys were depressed.

How to Program the Model 100 Keyboard

In this section we will develop a BASIC program to display the keyboard matrix on the liquid crystal display.

The Model 100 uses a nine column by eight row matrix for its keyboard (see Figure 6-1). Of the seventy-two possible positions in this matrix, seventy-one correspond to keys on the keyboard, leaving one position that does not correspond to any key.

The nine input lines are connected to eight bits of port B9h = 185d and one bit of port BAh = 186d (bit 0). A bit value of 1 turns off an input line and a bit value of 0 turns it on. This is just the opposite of what you might

expect, but the designers of the Model 100 placed "inverters" in the keyboard electronics that perform this reverse in logic.

The input ports are also used by the LCD and the clock. However, there is no conflict under normal operations because the keyboard output lines are never read when ports B9h = 185d and BAh = 186d are being used for other purposes.

The eight output lines are connected to port E8h = 232d for input to the CPU. This input port is not shared by any other part of the system, so there is never any confusion here.

For the output lines, a bit value of 1 means the corresponding switch was open (key not depressed), and a bit value of 0 means the switch was closed (key was depressed). Again, the logic is reversed relative to what you might expect, but it is consistent with the logic for the input.

	0	1	2	3	4	5	6	7	8
7	L	K	I	? /	* 8	↓	ENTER	F8	BREAK PAUSE
6	M	J	U	> .	& 7	↑	PRINT	F7	
5	N	H	Y	< ,	^ 6	→	LABEL	F6	CAPS LOCK
4	B	G	T	" '	% 5	←	PASTE	F5	NUM
3	V	F	R	: ;	$ 4	+ =	ESC	F4	CODE
2	C	D	E	] [	# 3	— -	TAB	F3	GRPH
1	X	S	W	P	@ 2	) 0	DEL BKSP	F2	CTRL
0	Z	A	Q	O	! 1	(9	SPACE	F1	SHIFT

Figure 6-1. The Model 100 keyboard matrix

The following program interactively displays the keyboard matrix for the Model 100. When you run this program, you will see a small rectangular display of pixels in the middle of your screen. If you now press various combinations of keys, you will see the corresponding pixels change from dark to light. To stop the program, just hold down CTRL C or BREAK for a moment.

```
100 ' DISPLAY KEYBOARD MATRIX
110 '
120   CLS
130   PRINT TAB(10);"KEYBOARD MATRIX"
140 '
150 ' MAIN LOOP
160 '
170 ' TURN OFF BACKGROUND TASK
180   CALL 30300
190 '
200 ' SET BIT 0 OF PORT BAh
210   X=INP(186)
220   OUT 186,X OR 1
230 '
240 ' SCAN THROUGH 8 COLUMNS
250   OUT 185,254:A0=INP(232)
260   OUT 185,253:A1=INP(232)
270   OUT 185,251:A2=INP(232)
280   OUT 185,247:A3=INP(232)
290   OUT 185,239:A4=INP(232)
300   OUT 185,223:A5=INP(232)
310   OUT 185,191:A6=INP(232)
320   OUT 185,127:A7=INP(232)
330 '
340 ' CHECK NINTH ROW
350 '
360 ' TURN OFF LOWER COLUMNS
370   OUT 185,255
380 '
390 ' TURN ON JUST THE NINTH
400   X=INP(186)
410   OUT 186,X AND 252
420   A8=INP(232)
430 '
440 ' DISPLAY THE MATRIX ON THE LCD
450 '
460 ' JUST ONE LCD DRIVER
470   OUT 185,128
480 '
490 ' SET UP LCD POSITION
```

```
500   OUT 254,0
510 '
520 ' SEND THE BYTES TO LCD
530   OUT 255,A8
540   OUT 255,A7
550   OUT 255,A6
560   OUT 255,A5
570   OUT 255,A4
580   OUT 255,A3
590   OUT 255,A2
600   OUT 255,A1
610   OUT 255,A0
620 '
630 ' ALLOW FOR A BREAK
640   PRINT CHR$(11);
650 '
660 ' GO BACK FOR MORE
670   GOTO 150
```

Let's look at the code in detail. Except for some initialization consisting of clearing the screen (line 120) and printing the title "KEYBOARD MATRIX" (line 130), the program consists of a loop (lines 150-670) that continually reads the columns of keyboard matrix and sends them to the LCD.

At the beginning of the loop (lines 170-180) the background task is turned off by calling the machine-language routine at 765Ch = 30,300d (see Chapter 4 for description).

In the next part of the loop, the columns are turned on one by one, and the keyboard output port (port E8h = 232d) is read each time into one of the variables A0 through A7. The bytes are not sent directly to the display because the liquid crystal display also uses ports B9h = 185d and BAh = 186d.

Notice how the ninth column is handled. This column is not controlled by port B9h = 185d as are the other columns. Instead, it is controlled by bit 0 of port BAh = 186d. Before the other columns are handled, this bit is set to 1, turning off this column. This has to be done carefully because this bit is on a port (port BAh = 186d) that is shared by other devices, including the power for the machine. You can see how carefully we set this bit by looking at lines 210-220. On line 210 the port is read, and then on line 220 its contents are ORed with 1 as they are put back. It should be clear that this changes only the one bit.

Next, the first eight columns are scanned (lines 250-320). In each case, one bit of port B9h = 185d is set to zero while the others are set to one, and port E8h = 232d is read into the appropriate variable: A0 through A7.

After all the other columns have been scanned, it's time to scan the last column. Again, port BAh = 186d is read before it is written to. Notice that both bits 0 and 1 of port BAh = 186d are made 0 when the port is set. This has two effects: it turns on the last column of the keyboard matrix (with bit 0), and it also disables the last two LCD drivers (controlled by bits 0 and 1). These are used for subsequent display of the matrix. The byte from this column is then picked up and stored in the variable A8.

By line 420 all the raw data from the keyboard are stored in variables A0 through A8, ready to be sent to an LCD driver for display on the screen. But before this data is sent, all but one of the LCD drivers are disabled (line 470), and the starting byte address is sent to that driver through port FEh = 254d. The matrix is then displayed by sending this raw data directly to a LCD driver through port FFh = 255d (lines 530-610). (See Chapter 4 for detailed description of how to program the LCD.)

Before the bottom of the loop there is a command that homes the cursor. Its purpose is to turn on the background task so that a CTRL C can be sensed if the user wishes to terminate the program. Any time you use the PRINT command, the background task is turned on.

Besides illustrating how the keyboard works, this program demonstrates how the LCD can be programmed directly to display complex data in real time.

Descriptions of ROM Routines

The ROM routines for the keyboard fall into three main classes: BASIC interrupt, background scanning task, and keyboard input. With the information presented here you should be able to write your own BASIC or machine-language programs to detect and handle any kind of regular or special combinations of keys. You can set the keyboard up for all sorts of input configurations. As with the case of special programming for the LCD screen, this is especially useful for games and educational software.

The ON KEY Interrupt Routines

Let's start with the interrupt routines for the eight function keys. There are four such commands: ON KEY GOSUB, KEY ON, KEY OFF, and KEY STOP. These provide standard ways to have specially programmed keys.

The code for ON KEY GOSUB is located at A5Bh = 2651d (see box). This code is also shared by the ON TIME$ GOSUB, ON COM GOSUB, and ON MDM GOSUB commands. Here, a routine at 1AFCh = 6908d is called to determine which of these different device types is required. For

the ON KEY interrupt, this routine returns with a value of 0208h in the BC register pair, that is, a value of 2 in the B register and a value of 8 in the C register. This indicates where and how much information for this type of interrupt will be stored in the system's interrupt table, which is located at F944h = 63,812d (see Figure 6-2).

Routine: ON...INTERRUPT/GOSUB — BASIC Command

Purpose: To initialize BASIC interrupts

Entry Point: A5Bh = 2651d

Input: Upon entry, the HL register pair points to the end of an ON...GOSUB command line.

Output: When the routine returns, the location of the subroutine to handle the particular interrupt is set.

BASIC Example:

```
CALL 2651,0,H
```

where H is the address of the end of an ON...GOSUB command line.

Special Comments: None

For the ON KEY GOSUB command, the value 8 in the C register indicates that a maximum of eight locations (one for each function key) will be used, and the 2 in the B register indicates that they will start on the third location of this table (the first two are occupied by the TIME$ and the MDM or COM interrupts). (Note that the MDM and COM have exactly the same interrupt locations.) Each location in this table has three bytes, one for status and two for the location of the interrupt subroutine. The last part of the routine at A5Bh = 2651d reads the list of BASIC subroutines from the end of the ON KEY GOSUB command line and places their locations into the proper slots of the interrupt table (see Figure 6-2).

The KEY ON, KEY OFF, and KEY STOP use the same code and work in the same way as the TIME$ ON, TIME$ OFF, and TIME$ STOP commands described in Chapter 5. That is, they cause transitions of finite state machines (see Chapter 5) whose states are stored as status bytes in the interrupt table. Each function key has its own status byte and hence its own finite state machine.

The following program lets you "peek" at these status bytes, graphically demonstrating how these interrupt functions work. You can watch the state of the F4 interrupt as you turn it on, off, and stop it. The keys F1, F2, F3 have been assigned to do these tasks for the function key F4 so that you can have direct and convenient control over these functions for one particular interrupt. As you press these keys you will see the fourth status byte change. You should review the discussion of BASIC interrupts in Chapter 5 to see what the various values mean.

```
100 ' DISPLAY INTERRUPT STATES
110 '
120   CLS
130   PRINT "FUNCTION KEY STATES"
140   PRINT:PRINT:PRINT
150   PRINT "F1 = KEY(4) ON"
160   PRINT "F2 = KEY(4) OFF"
170   PRINT "F3 = KEY(4) STOP"
180 '
190 ' TURN ON INTERRUPTS
200   ON KEY GOSUB 320,350,380,410
210   KEY(1) ON:KEY(2) ON:KEY(3) ON
220 '
230 ' PRINT ON THIRD LINE
240   CALL 17020,,2
250 '
```

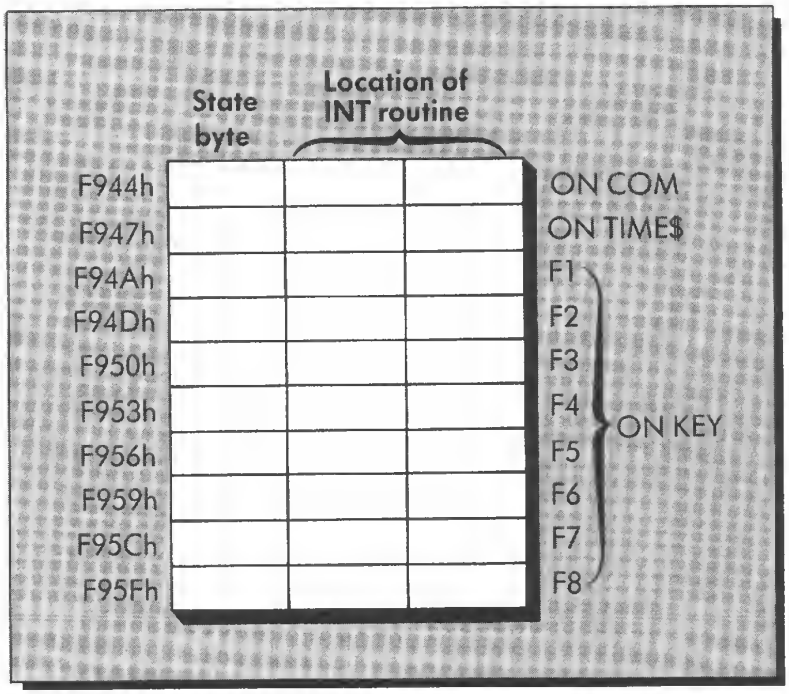


Figure 6-2. The system interrupt table

```
260 ' DISPLAY THE STATUS OF THE KEYS
270   FOR I = 63818! TO 63841! STEP 3
280   PRINT USING "####";PEEK(I);
290   NEXT I
300   GOTO 230
310 '
320 ' KEY 1 INTERRUPT ROUTINE
330   KEY(4) ON
340   RETURN
350 ' KEY 2 INTERRUPT ROUTINE
360   KEY(4) OFF
370   RETURN
380 ' KEY 3 INTERRUPT ROUTINE
390   KEY(4) STOP
400   RETURN
410 ' KEY 4 INTERRUPT ROUTINE
420   RETURN
```

Let's look at the code for this program. In lines 120-170 the background display is set up, showing a title and a description of the function keys. In lines 190-210 the locations of ON KEY interrupt routines are defined and the interrupts for F1, F2, and F3 are turned on.

Lines 230-300 form the main loop, which repeatedly displays the status bytes. At the top of this loop, the cursor is placed at the beginning of the third line by CALLing the cursor positioning routine at 427Ch = 17,020d (see Chapter 4). This routine expects the row position in the L register and the column position in the H register. In each case, the numbering starts at zero. The CALL command expects the value of the HL register in its third parameter. In this case we pass a value of 2, causing zero to be in H and 2 to be in L, thus setting the cursor to row 2, column 0.

In the last part of this loop, a short FOR loop (lines 270-290) prints the status bytes on the screen. They appear on the third line of the screen, where we just placed the cursor.

After the main loop come our BASIC interrupt subroutines for F1 through F4. As we have already observed, the first three keys control the interrupt for the fourth. The routine for F1 is a command to turn on the F4 interrupt, the routine for F2 is a command to turn off the F4 interrupt, and the routine for F3 is a command to stop the F4 interrupt.

The Clock-Cursor-Keyboard Background Task

The keyboard section of the background task performs an essential function for the Model 100 by constantly scanning the keyboard. It detects

what keys are hit and puts their character codes into a keyboard character buffer.

The code for the keyboard section of the background task starts at 7055h = 28,757d and runs to about 7241h = 29,249d. It is divided into four main subsections: general management, key detection, key decoding, and character buffer management (see box).

Routine: Keyboard Scanning

Purpose: To scan the keyboard as part of the background task

Entry Point: 7055h = 28,757d

Input: Monitors the keyboard

Output: Puts codes for the keys in the keyboard input buffer

BASIC Example: Not applicable

Special Comments: Not a callable routine — part of background task

Management

The general management subsection controls program flow and timing (see box). It sets up a "graceful" exit from the keyboard section, and it times the keyboard scanning by allowing it to be performed only every third time that the background task is executed.

Routine: Management of Keyboard Scanning

Purpose: To set up timing and exit conditions for the keyboard scanning background task

Entry Point: 7055h = 28,757d

Input: A counter located at FF8Fh

Output: The counter is decremented if not already equal to one. If it is one, the counter is set to a value of three.

BASIC Example: Not applicable

Special Comments: Not a callable routine — part of the background task

The "graceful" exit from this section is accomplished by pushing the address 71F4h = 29,172d onto the stack. This address points to the very last part of the background task; hence any RETurn instruction in this section will automatically cause the CPU to jump there. The code at this "finishing" address of 71F4h = 29,172d has the correct series of "POPs" and RETurn to make a proper exit from the background task.

The waltz-like "every third time" timing is controlled by a countdown counter located at FF8Fh = 65,423d. This counter is loaded with 3 every time it reaches zero. Since keyboard scanning is performed only every third time that the background task is executed, this occurs about every 12 milliseconds (recall that the background task itself is performed about every 4 milliseconds.)

Key Detection

The key detection subsection must determine and record the matrix positions of keys that are pressed down on the keyboard. You can use the ideas described here to develop your own detection routines. For example, you could write a program that detects certain double and triple key combinations used with programs that simulate various instruments or machines.

The routine for key detection begins at 7060h = 28,768d (see box). It contains a short block of code starting at 7066h = 28,774d, which decodes the ninth column of the keyboard matrix, and then a loop (starting at 7080h = 28,800d) for detecting all the other keys. The loop is executed eight times, once for each remaining column of the keyboard matrix. The columns are scanned in reverse order, from last to first.

Routine: Key Detection

Purpose: To determine which keys have been hit

Entry Point: 7060h = 28,768d

Input: Monitors the keyboard

Output: Special tables in RAM contain information about the keyboard matrix.

BASIC Example: Not applicable

Special Comments: Not a callable routine — part of the background task

The job of key detection is complicated by the fact that the various shift keys ([SHIFT], [CTRL], [GRPH], [CODE], [NUM], and [CAPS LOCK]) are used to modify the meaning of regular keys. For this reason and others, detecting the shift keys differs from detecting other keys.

The shift keys are all in the last (ninth) column of the keyboard matrix, which is controlled by bit 0 of port BAh = 186d. With the exception of the [BREAK/PAUSE] key, which is also on this column, all keys are controlled by the eight bits of port B9h = 185d. The ninth column is scanned first, in a separate section from the other columns.

The first few instructions of the keyboard detection routine set up pointers to two buffers, each nine bytes long. In each buffer there is one byte for each column of the keyboard matrix. The first buffer runs from FF91h = 65,425d to FF99h = 65,433d, and the second runs from FF9Ah = 65,434d to FFA2h = 65,442d (see Figure 6-3). The ending address of the first buffer is placed in HL, and the ending address of the second is placed in DE. They are placed this way because the columns of the keyboard matrix are scanned backward from last to first. This order is convenient because it allows the shift keys to be scanned first and it makes the loop counter (B register) equal to the number of the column.

Here is a short program that displays these buffers on the LCD screen. As you type characters, you will see the contents of the buffers change.

```
100 ' KEYBOARD MATRIX BUFFER DISPLAY
110 '
120   PRINT CHR$(11);
130   FOR I = 65425 TO 65433
140    PRINT PEEK(I);
150   NEXT I
160   PRINT
170   FOR I = 65434 TO 65442
180    PRINT PEEK(I);
190   NEXT I
200   GOTO 120
```

This program consists of an infinite loop that displays the keyboard matrix buffers over and over again. At the top of the loop (line 120), the cursor is placed in home position. Lines 130-160 display the first buffer on the top line of the display, and lines 170-190 display the second buffer on the second line of the display.

Two buffers are needed for the keyboard matrix in order to help distinguish among the following conditions: 1) a key has not been held down long enough to register; 2) a key has been just hit, but this is the first keyboard scan that detects it; 3) a key is depressed but has already been detected

during a previous scan; and 4) a key has been depressed long enough to activate the repeat feature.

In general, a key is detected in a two-step process, one step for each buffer. In the first step, the byte for the key column is picked up from the keyboard output port and compared with the current value in the first buffer, and then the current value in the first buffer is replaced by this new value. If the new and old values for the first buffer disagree, this must be the first scan to detect this change in the keyboard. In this case, no further action is taken for this particular column.

If the new and old values for the first buffer agree, the second step is performed. In this step the corresponding byte in the second buffer is checked and updated if it is different. This time no further action is taken if the values agree, because this means that the key has already been detected and should not be counted another time. However, if the values differ, the position of the key in the matrix is computed, ready for the decoding stage.

The key detection process produces several bytes of data that are used by the decoding process. Primary among these are FFA3h = 65,443d, which

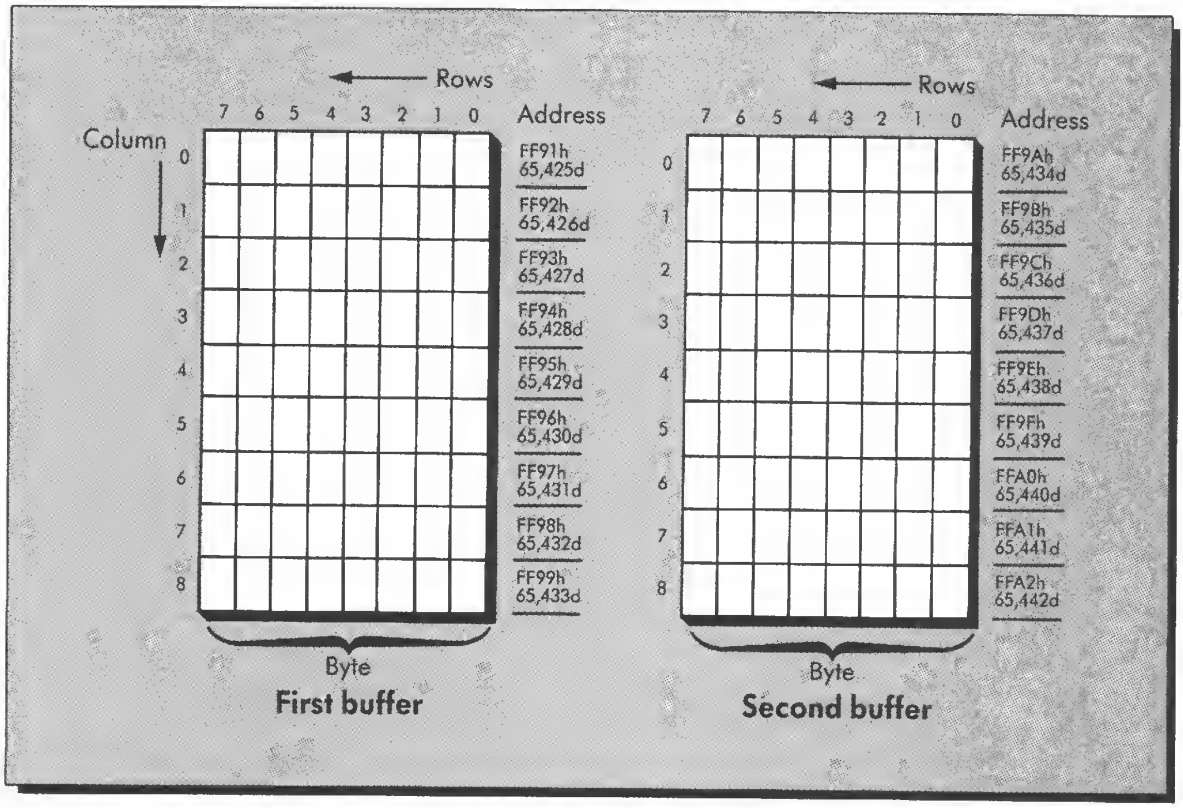


Figure 6-3. Keyboard matrix buffers

contains the shift code byte (for column nine), and FFA6h = 65,446d, which contains the position of the key in the matrix. This last value is obtained by multiplying the column position by eight and adding in the row position. The code for this calculation starts at 70F0h = 28,912d.

The repeat feature for keys is handled in the code starting at about 70B0h = 28,848d. The length of the keyboard buffer is first checked, for if there is more than one character in the buffer, the repeat feature will not be active. A counter at FFA4h = 65,444d counts down 84 cycles of keyboard scanning before the repeat feature is activated. Using the 12-millisecond cycle time, this gives about a one second delay before the key begins to repeat automatically. Once the key begins to repeat, for each cycle of the repeat, the counter is set back to six (at 70BBh = 28,859d), giving a frequency of about fourteen repeats per second. (This checks with the timing results we measured with a stopwatch.)

Key Decoding

In the decoding section the key positions are converted into ASCII codes. You can use the basic principles of this routine to convert key combinations to any sort of code that is required. The main job is in setting up the proper translation tables.

The decoding subsection begins at about 7122h = 28,962d (see box).

Routine: Key Decoding

Purpose: To convert key combinations into corresponding ASCII codes

Entry Point: 7122h = 28,962d

Input: Upon entry, the key position number is in location FFA6h = 65446d and the shift code (what shift keys have been depressed) is in location FFA3h = 65,443d.

Output: ASCII code for the corresponding character

BASIC Example: Not applicable

Special Comments: Not a callable routine — part of the background task

First the position of the key is retrieved from FFA6h = 65,446d. Figure 6-4 shows how the position numbers correspond to the labels on the keys.

The keys in the white area of this diagram are handled first. They have position numbers 0 through 43 and correspond to all the alphabetical, numerical, and punctuation keys. Not included are the keys around the outside of the keyboard, such as (ESC), (TAB), (CTRL), (SHIFT), (SPACE), (ENTER), and the function keys.

The keys with positions 0 through 43 map to ASCII in six different ways, depending upon the shift code. There are six tables in memory to handle these six different cases (see Appendix D). Each table is forty-four bytes long.

The code that manages these tables extends from 7122h = 28,962d to 71B8h = 29,112d. In this code, the BC, HL, and DE are used to help compute the table look-up address. The BC register pair holds the key position, the HL register pair holds the base address for the particular table, and the DE register pair holds the value 44 or 0 for use with the (SHIFT) key. These are all added together to compute the look-up address for the ASCII code.

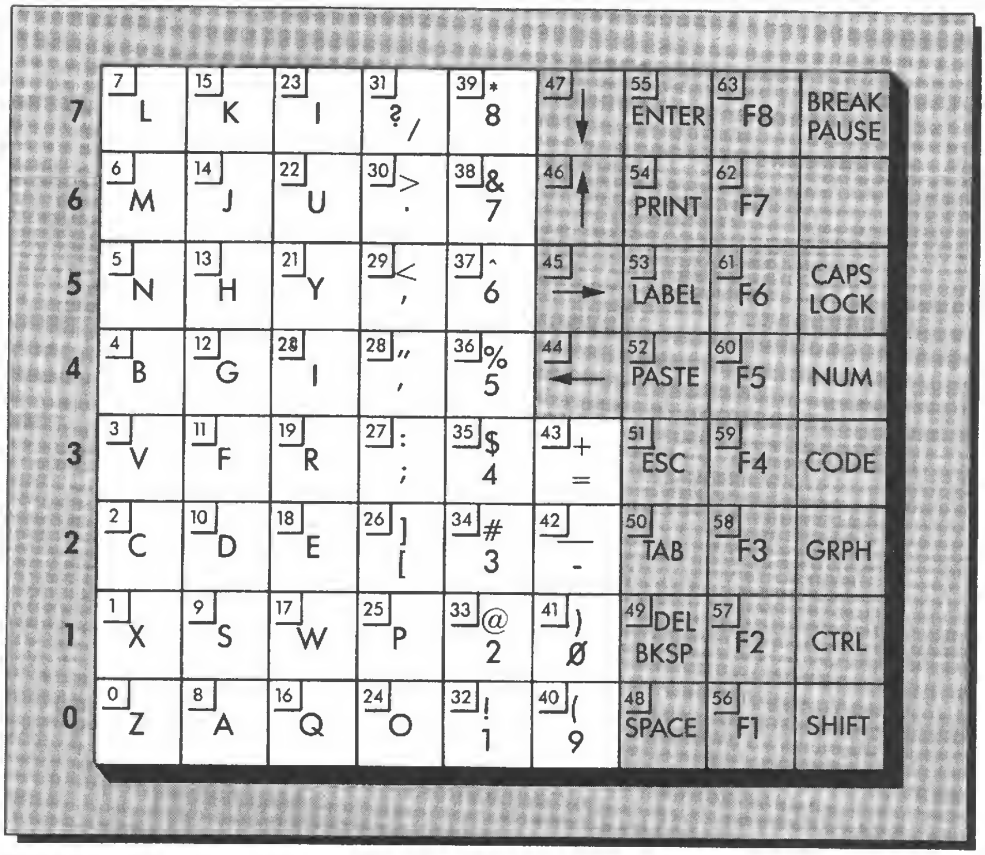


Figure 6-4. Keyboard matrix table

The first table starts at 7BF1h = 31,729d and contains the ASCII codes for the unshifted characters. This includes the lowercase letters, the numbers, and the unshifted punctuation mark keys. In this case HL is set equal to the value 7BF1h = 31,729d, and DE is made equal to zero for the look-up.

The second table starts at 7C1Dh = 31,773d and holds the ASCII codes for uppercase letters and shifted punctuation marks. In this case HL is loaded with 7BF1h = 31,729d as before, but DE has the value 2Ch = 44d. Thus the computed address will point into this second table. The CPU instructions at 7133h = 28,979d, 7136h = 28,982d, 7185h = 29,061d, and 7188h = 29,064d examine the [SHIFT] key bit and set DE accordingly.

The third table starts at 7C49h = 31,817d and holds ASCII codes generated while holding down the [GRPH] key, and the fourth table starts at 7C75h = 31,861d and holds ASCII codes for characters generated while holding down both the [GRPH] and the [SHIFT] keys. In both cases the HL register is loaded with the value 7C49h = 31,817d, but as above the DE register is used to "shift" between the two cases.

The fifth table starts at 7CA1h = 31,905d and holds ASCII codes generated while holding down the [CODE] key, and the sixth table starts at 7CCDh = 31,949d and holds ASCII codes for characters generated while holding down both the [CODE] and the [SHIFT] keys. Again, the HL register is loaded with a base value (this time 7CA1h = 31,905d), and the DE register is used to "shift" between the two cases.

There are other parts to the decoding routine to handle the [CTRL] key (at 720Ah = 29,194d), the [CAPS LOCK] key (at 722Ch = 29,228d), the [NUM] key (at 7233h = 29,235d), the function keys (at 715Bh = 29,019d), and the arrow keys (at 7222h = 29,218d). There are special tables in the ROM to handle some of these. If the [NUM] key is depressed, the table at 7CF9h = 31,993d (see Appendix P) is used to search for the values of the keys in the Model 100 keyboard's number pad area.

Two tables contain ASCII codes for the keys in positions 44 through 63 (see Appendix Q). The first table starts at 7D07h = 32,007d and contains ASCII codes for these keys when [SHIFT] is not depressed. The second table starts at 7D1Bh = 32,027d and contains ASCII codes for these keys when [SHIFT] is depressed. The routine for decoding these keys runs from 7148h = 29,000d to 7158h = 29,016d and from 7222h = 29,218d to 7229h = 29,225d.

Character Buffer Management

This section illustrates one way to build a buffer to receive characters from an I/O process. In Chapter 7, on serial communications devices, we will see another way.

In the buffer management subsection, starting at 71C4h = 29,124d, the ASCII codes for the keys are put into the keyboard input buffer (see box).

Routine: Keyboard Character Buffer Management

Purpose: To put key codes into the keyboard input buffer

Entry Points: 71C4h = 29124d, 71D5h = 29,141d, and 71E4h = 29,156d

Input: Upon entry, the ASCII code for the character is in the A register and/or C register (depending upon which entry point is used).

Output: The key codes are put in the keyboard input buffer.

BASIC Example: Not applicable

Special Comments: Not a callable routine — part of the background task

There are several entry points. The one at 71D5h = 29,141d is for placing characters in an empty buffer, and the one at 71E4h = 29,156d is for storing subsequent characters in the buffer.

The number of characters currently in the buffer is stored at location FFAAh = 65,450d. The buffer can store a maximum of thirty-two characters, and each character takes two bytes of storage. The buffer is a queue: That is, it is a list in which new entries are added to the end of the list and old ones are taken from the beginning.

For regular ASCII characters, the first byte contains the ASCII code and the second byte is zero. Certain other keys, namely the function keys and the [LABEL], [PRINT], [PRINT] with [SHIFT], and [PASTE], are stored with a numeric code in the first byte and a value of 255 in the second byte. The numeric codes for these keys are 0 through 11, respectively.

Here is a BASIC program that displays the keyboard input buffer. You can use it to watch this buffer as you hit various keys. Just run the program and watch the screen as you hit some keys. You should type slowly, because it takes a while for this BASIC program to make a full cycle. The first

number displayed is the number of characters currently in the buffer. After that every two bytes is where a character is stored in the buffer. You should compare the display on the screen with the foregoing description of the buffer.

```
100 ' KEYBOARD BUFFER
110 '
120   PRINT CHR$(11);
130   FOR I = 65450 TO 65514
140    PRINT USING "#####"; PEEK(I);
150   NEXT I
160   GOTO 120
```

On line 120 of this program, the cursor is put into the home position (upper left corner). On lines 130-150, the contents of the buffer are displayed. On line 160, the program loops back to line 120 to display the buffer again.

The Keyboard Input Routines

The keyboard input routines provide a way to grab characters from the keyboard character buffer. This is the "official" way to get characters from the keyboard. It is the only way that the rest of the Model 100's ROM routines can get characters from the keyboard.

The KYREAD Routine

There are several routines for getting characters from the buffer. For a few more than are presented here, see the *Model 100 ROM Functions* (700-2245) from Radio Shack.

Let's look first at a routine called KYREAD (see box), located at 7242h = 29,250d. It checks whether or not there is a character in the buffer. If there are no characters, it returns with the Z flag set (Z). If there is a character, it returns with the ASCII code for the key in the A register and the Z flag clear (NZ). However, if the C flag was also set, then the character corresponds to one of the special keys [F1] through [F8], [LABEL], [PRINT], [SHIFT][PRINT], and [PASTE]. In that case the A register contains a corresponding numeric code, 0 through 11.

Routine: KYREAD

Purpose: To read a key from the keyboard input buffer

Entry Point: 7242h = 29,250d

Input: From the keyboard

Output: When the routine returns, the Z flag indicates if there are any characters in the keyboard input buffer. If the Z flag is set (Z), there are no characters. If the Z flag is clear (NZ), the ASCII code of the next character is in the A register. If the C register is also set (C), then the character corresponds to one of the special keys F1 through F8, LABEL, PRINT, SHIFT PRINT, and PASTE. In that case the A register contains a corresponding numeric code, 0 through 11.

BASIC Example: Not applicable

Special Comments: None

The routine first turns off the background task and then checks the variable at FFAAh = 65,450d, which holds the number of characters in the buffer. If this is zero, then it returns, turning the background task back on.

If there is at least one character in the buffer, then its code is loaded into a register, and the other entries in the buffer are moved one character position toward the front of the buffer. Then the routine returns, turning on the background task.

The BRKCHK Routine

The next routine is called BRKCHK (see box). It checks for a break or pause character (CTRL C or CTRL S). These are also generated by the BREAK PAUSE keys. If a break character was hit, it returns with the zero flag clear (NZ) and the ASCII code for the break or pause character in the A register; otherwise it returns with the zero flag set (Z).

Routine: BRKCHK

Purpose: To check for break or pause character

Entry Point: 7283h = 29,315d

Input: From the keyboard

Output: If a break character was hit, it returns with the zero flag clear (NZ) and the ASCII code for the break or pause character in the A register; otherwise it returns with the zero flag set (Z).

BASIC Example: Not applicable

Special Comments: None

The BRKCHK routine is located at 7283h = 29,315d. It first checks location FFEBh = 65,515d. This location is set by the key detection routines. It is normally zero, but if a break or pause character is detected, the ASCII code for that character is stored in this location. Special codes with the eighth bit turned on are also stored here when a function key is detected.

The BRKCHK routine clears location FFEBh = 65,515d after checking it. It first checks the eighth bit to see if a function key has been detected. If this is nonzero, a function key must have been detected, and it processes an interrupt for that key. If the eighth bit is clear (zero), then the routine returns with the zero flag set according to the contents of location FFEBh = 65,515d (nonzero or zero according to whether or not a break or pause was detected).

The KEYX Routine

The last input routine we'll look at in this chapter is called KEYX (see box). It checks the keyboard queue for normal characters or a break. It does not actually return any characters, only CPU flags. If there is at least one character in the buffer, it returns with the Z flag clear (NZ); otherwise the zero flag is set (Z). If a break was hit, then the carry is set (C); otherwise it is clear (NC).

Routine: KEYX

Purpose: To check for a character from the keyboard

Entry Point: 7270h = 29,296d

Input: From the keyboard

Output: If there is at least one character in the buffer, the routine returns with the Z flag clear (NZ); otherwise the zero flag is set (Z). If a break was hit, then the carry is set (C); otherwise it is clear (NC).

BASIC Example: Not applicable

Special Comments: None

The routine is located at 7270h = 29,296d. It first calls the BRKCHK routine (described previously). If there is no break or pause character, it checks location FFAAh = 65,450d to see if there are any characters in the buffer, returning with the zero flag set accordingly. If there was a break or pause, it checks the ASCII code in FFEBh = 65,515d for a break (ASCII 3), as opposed to a pause (ASCII 13h = 19d). If indeed there was a break, the routine sets the carry and returns; otherwise it checks to see if the buffer has "pause" characters pending.

Summary

In this chapter we have explored the Model 100's keyboard. We have seen how to scan it using BASIC and how the ROM routines in the Model 100 interface between the keyboard and the rest of the computer. We have seen that the actual hardware keyboard is fairly simple, but the software is complicated by the fact that the SHIFT, CTRL, BREAK and PAUSE keys have to be handled in special ways.

7

Hidden Powers of the Communications Devices

Concepts
- How an RS-232 serial communications line works
- The RS-232 connector and the modem

ROM Routines for the Communications Devices
- BASIC interrupt commands
- Initializing and shutting down the UART
- Dialing the telephone
- Reading from the serial communications line
- Writing to the serial communications line

The serial communications line provides a vital link between the Model 100 and other computers, allowing it to become part of a larger information handling system. Through the serial communications line, the Model 100 can be used as a terminal for other computers and as a detachable unit that can be used to carry files from a larger computer to places where larger computers are unavailable. For example, you can download a file into the Model 100 at work, edit it at home, and then bring it back to the main computer the next day. Through the modem connection, you can also connect to other computers over telephone lines.

In this chapter we will explore the secrets of the Model 100's modem and RS-232 serial communications devices. We will see how these devices work and how they are set up in the Model 100. Then we will study the ROM routines that control them.

How an RS-232 Serial Communications Line Works

There are two major ways to send computer data: parallel and serial. Parallel transmission is normally used within the computer and for short distances outside the computer such as between the computer and a printer. Serial transmission is the rule for longer distances such as over the phone lines or from one building to another.

With parallel transmission, all the bits of a byte of data are transmitted at the same time, each over its own separate signal line. With serial transmission, in contrast, the bits are sent one at a time over a single signal line (see Figure 7-1).

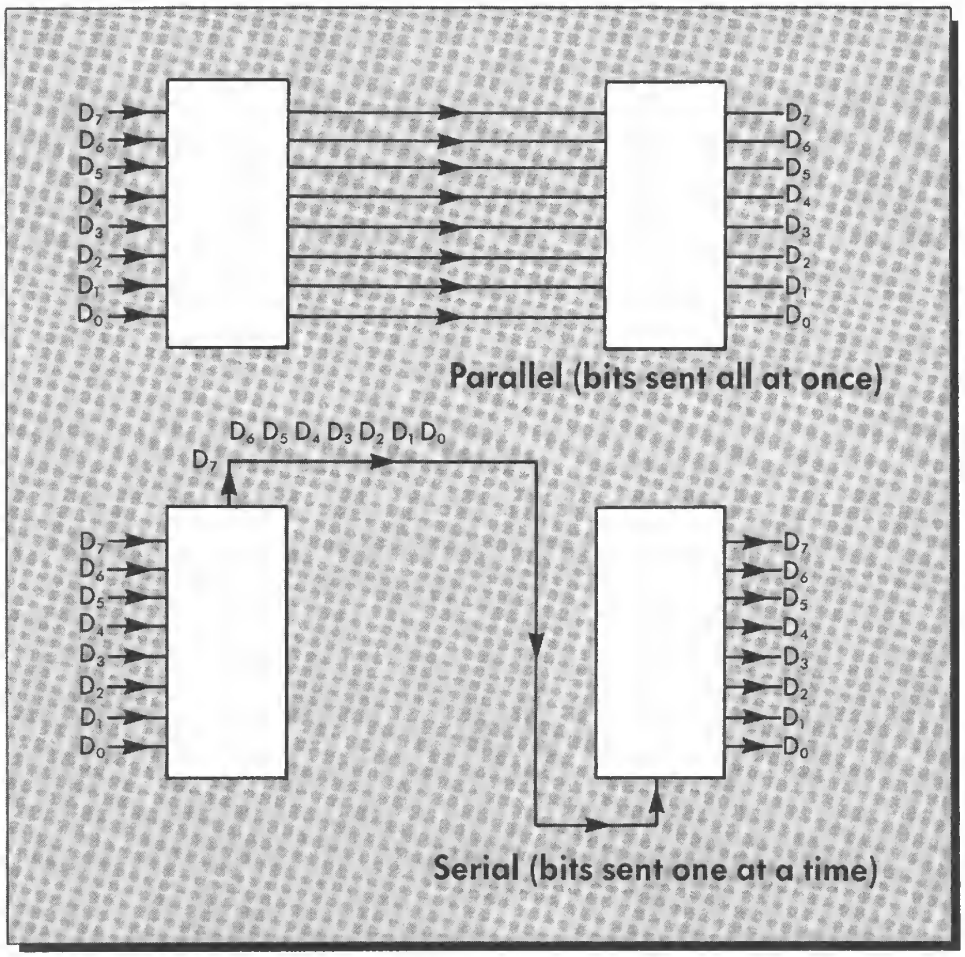


Figure 7-1. Parallel and serial transmission

Advantages of Serial Transmission

The main advantages of serial transmission are that fewer wires are needed and standards for this type of transmission are well established. The main disadvantages are that special hardware is needed to convert between the computer's internal parallel format and the external serial format and that serial transmission is limited to slower speeds than parallel transmission.

Let's look in more detail at the advantages, starting with the fact that fewer wires are needed for serial transmission. For a two-way serial communications line, a minimum of three signal lines (wires) is needed: one for each of the two directions and one as a ground line. Other signals can be added to provide hardware control of the flow of information. The serial communications port on the Model 100 contains six different signal lines plus a ground, but it's quite possible to use only three of these for many applications (see Figure 7-2).

The second advantage of serial transmission is its well-established standards. Standards increase the transportability of data and program files and thus increase overall productivity. The main standard for serial communications is called RS-232C; this is what the Model 100 uses. The RS-232C standard encompasses a number of different speeds (called baud rates) and a number of different formats (see Figure 7-3). When you set up your communications on the Model 100 you have to select the baud rate, word length, parity, and number of stop bits. Once these have been properly selected, you should be able to communicate with any other computer that has a RS-232C serial line with the same choices of speed and format. Since most computer systems have the option of communicating in this way, your Model 100 computer can communicate to most other systems.

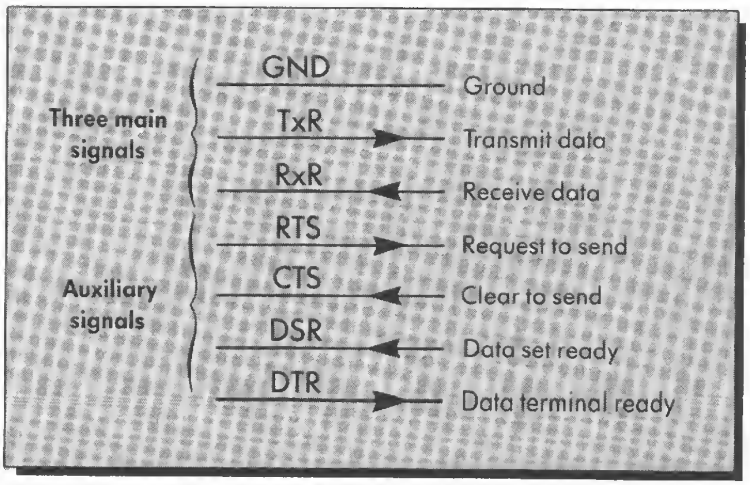


Figure 7-2. Signals for serial transmission

Disadvantages of Serial Transmission

Now let's look at the disadvantages of serial communications. The first is the extra hardware required. Fortunately, this hardware has been neatly packaged in computer chips called UARTs. This stands for *Universal Asynchronous Receiver Transmitter*. The word "Asynchronous" refers to the fact that bytes of data can be sent one by one as they become available, instead of in large blocks as with "synchronous" transmissions. There are a number of different UARTS available today. The Model 100 uses a UART chip called an IM6402.

The second disadvantage of serial transmission is speed. The difficulty is that the bits are sent one at a time. When a byte is to be sent, it is loaded into the UART in parallel. Then the UART peels off the bits, one by one, to be sent over the data signal line (see Figure 7-4). Sending one byte of information requires that a series of about ten bits be sent over the serial communications line. The extra bits are used to separate the bytes from each other. The opposite process happens when data are received. In this

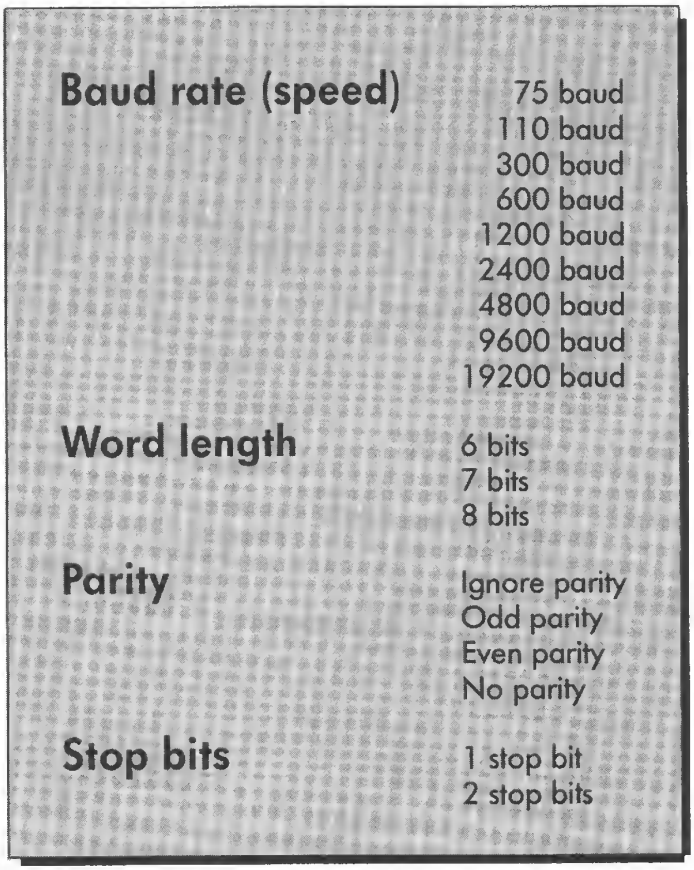


Figure 7-3. Typical speeds and formats for serial transmission

case, the bits from the serial communications line accumulate in the UART until a byte is complete and then are transferred in parallel into the computer (see Figure 7-4). Special status lines from the UART to the computer indicate when each of these processes is complete and the UART is able to handle more characters.

Standard speeds for serial transmission range from 75 to 19,200 baud (bits per second) or 7.5 to 1920 bytes per second. Data bytes are transmitted inside the computer in parallel at speeds of up to several million bytes per second, but transmission between computers at such high speeds simply is not practical. The Model 100 can handle the full range of baud rates, but not without problems. For example, the LCD screen can display characters at a rate of only about 90 characters per second, and the practical limit for downloading files (reading them into the Model 100's memory) is about 240 characters per second.

To overcome differences in transmission speeds between devices, serial transmission lines send signals back and forth to control the flow of data. The Model 100 uses what is called an XON/XOFF protocol. With this method, the receiving device sends a CTRL S (XOFF) when it can no longer safely handle more data from the transmitting device. It sends a CTRL Q

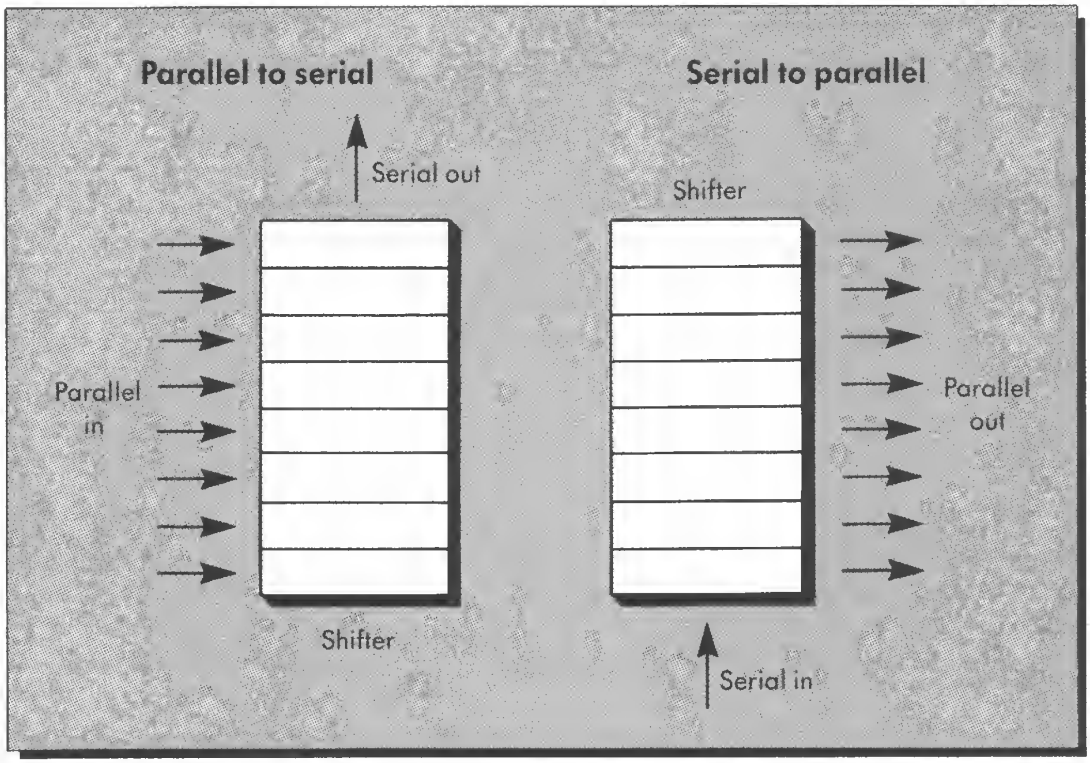


Figure 7-4. Converting from parallel to serial and from serial to parallel

(XON) when it can handle more. Later in the chapter we will study the ROM routines in the Model 100 that handle this protocol.

The RS-232C Connector and the Modem

The UART in the Model 100 is connected to the outside world in two ways: through the RS-232C connector and through the modem. The RS-232C connector provides a standard way of connecting two devices directly, and the modem converts back and forth between the serial bits of data of the UART and audio signals suitable for transmission over the telephone lines (see Figure 7-5).

Bit 3 of port BAh = 186d switches between the RS-232C connector and the modem. When this bit is 0, the UART is connected to the RS-232C connector, and when this bit is 1, the UART is connected to the modem (see Figure 7-6).

The modem and the RS-232C connector behave much the same to a programmer. There are, however, several differences. The modem has the automatic dial function, and the RS-232C connection has some extra status and control signals. As we shall see, dialing is done by taking the telephone rapidly on and off its hook. The extra signals for the RS-232C connector are RTS (Ready to Send), DTR (Data Terminal Ready), CTS (Clear to Send), and DSR (Data Set Ready). The Model 100 is set up as a data terminal; thus, the first two signal lines are output, and the second two are input.

ROM Routines for the Communications Devices

The ROM routines for the communications devices perform such tasks as handling BASIC interrupts, initializing the UART, switching between the modem and RS-232C connector, reading from and writing to the UART, and dialing the phone.

The BASIC Interrupt Commands

BASIC has interrupt commands for the communications line that are similar to those for the clock and the keyboard. The interrupt control commands are ON MDM GOSUB, ON COM GOSUB, MDM ON, COM ON, MDM OFF, COM OFF, MDM STOP, and COM STOP. The keyword MDM refers to the modem, and the keyword COM refers to the communications line in general, but, the same routines are used to handle both cases. This is because the interrupts happen in the UART, which lies between the computer and both these methods of transmitting serial data.

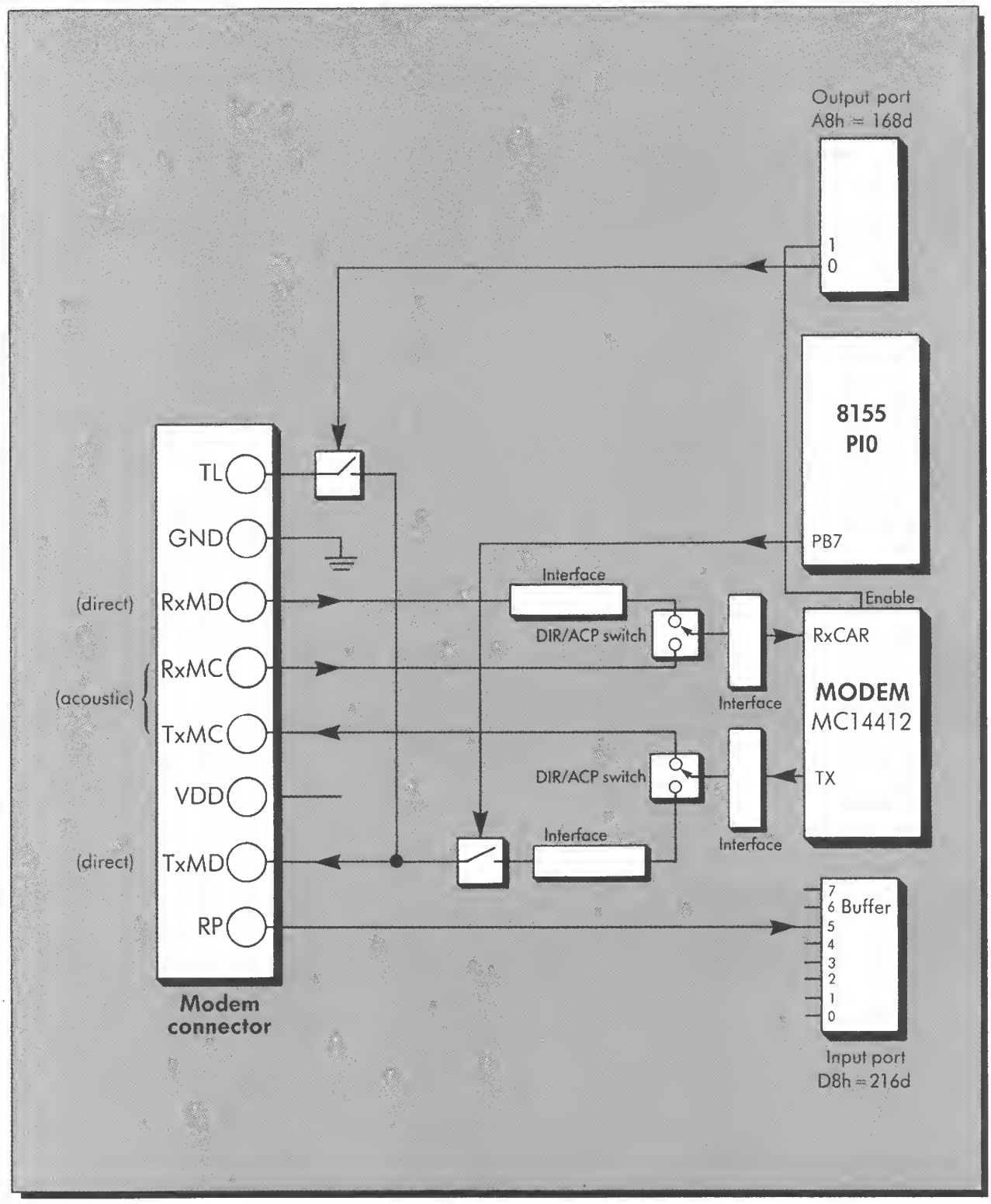


Figure 7-5. The modem

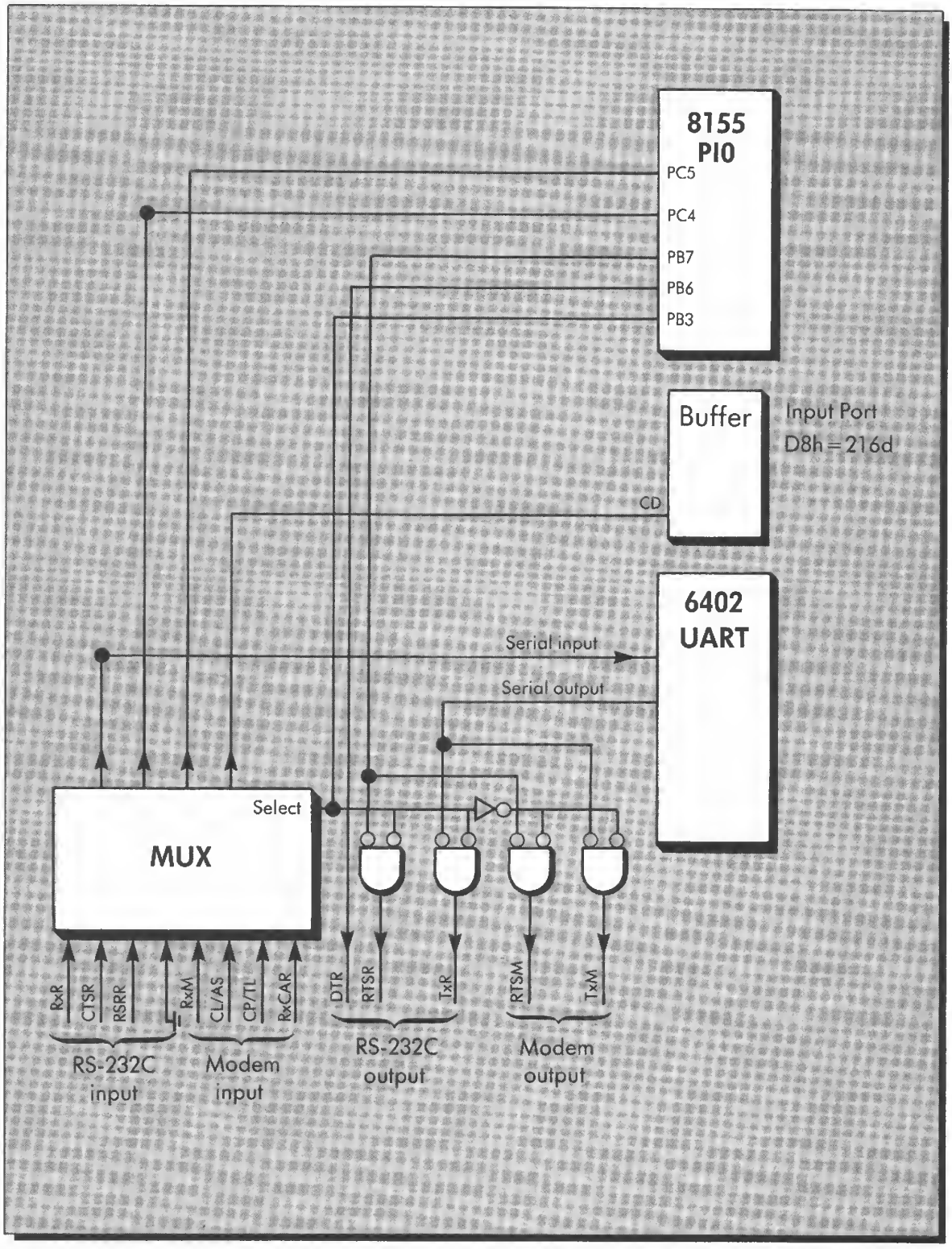


Figure 7-6. Switching between the modem and the RS-232C

In Chapters 5 and 6 we discussed how BASIC interrupt routines operate on the BASIC interrupt table (starting at F944h = 63,812d). The first three bytes of this table are reserved for the COM or MDM interrupt and contain the interrupt status word and the location of the BASIC interrupt subroutine for the communications devices.

Initializing and Shutting Down the UART

The routine to initialize the UART is called INZCOM and is located at 6EA6h = 28,326d (see box). This routine will help you to write your own code to set up the UART any way you want.

Routine: INZCOM

Purpose: To initialize the UART that controls the communications line

Entry Point: 6EA6h = 28,326d

Input: Upon entry, it expects the H register to specify the baud rate (1 through 9 as used in the STAT program), the L register to contain the UART configuration code, and the carry flag to indicate whether the modem or the RS-232C connector is to be used (set if RS-232C and clear if modem). In the configuration code, bit 0 specifies the number of stop bits (0 = 1 stop bit and 1 = 2 stop bits), bits 1 and 2 specify the parity (00 = none, 01 = even, and 10 = odd), and bits 3 and 4 specify the word length (00 = 6, 01 = 7, and 10 = 8 bits).

Output: When the routine returns, the UART is properly initialized.

BASIC Example:

```
CALL 28326,0,H
```

where H is a 16-bit number that contains the configuration information as specified above.

Special Comments: This routine will not update the configuration information that is stored in memory; thus, it is not permanent.

If the carry flag is clear, the modem was selected. In this case, the routine loads a 3 into the H register and 2Dh = 45d into the B register. This sets the baud rate to 300 and turns the modem on. Then it disables interrupts using the DI command and calls a routine called BAUDST to set the baud rate.

The BAUDST routine is located at 6E75h = 28,277d (see box). It expects the H register to contain a number from 1 to 9 that specifies the baud rate, as described above. This routine uses a table located at 6E94h = 28,308d to look up the correct patterns to send to ports BCh = 188d and BDh = 189d. These ports program a timer in the 8155 PIO chip to produce a square wave, which is sent to the UART to control the baud rate.

Routine: BAUDST

Purpose: To set the baud rate for the serial communications line

Entry Point: 6E75h = 28277d

Input: Upon entry, the H register contains a number from 1 to 9 that specifies the baud rate, using the same correspondence as is used by the STAT program.

Output: When the routine returns, the baud rate is set as specified.

BASIC Example:

```
CALL 28277,0,H
```

where H is 256 times (because it's the H register) the baud rate number decribed above.

Special Comments: None

Bits 6 and 7 of port BDh = 189d are always set to 01. This specifies that the output of the timer is a continuous square wave. The other possibilities are 00 for a single cycle of square wave, 10 for a single pulse, and 11 for continuous pulses. The other bits of these ports form a fourteen-bit binary number. Bits 0 through 7 of port BCh = 188d form the lower eight bits, and bits 0 through 5 of port BDh = 189d form the upper six bits. In a moment we will explain how this number helps to determine the baud rate, but first, let's trace the clock signals that control the baud rate.

The timing signal for the baud rate originates with the 4,915,200 cycles per second crystal that runs the CPU and provides the timing for the main circuits of the computer. The CPU divides this frequency in half and sends it on to drive other devices such as the timer in the 8155 PIO. This timer further divides the frequency by the fourteen-bit binary number previously referred to. The result is passed to provide a baud rate clock for both the receiver and the transmitter circuits in the UART. The actual baud rate produced by the UART is one sixteenth of the frequency of this last signal.

For example, if you select 300 baud, the fourteen-bit binary number will have a value of 512 in the table. The baud rate will be 4,915,200 divided by 2, divided by 512, divided by 16, which is exactly 300.

After ports BCh = 188d and BDh = 189d are programmed, port B8h = 184d is set to a value of C3h = 195d. The upper two bits of this byte start the timer, and the lower six bits define how ports B9h = 185d, BAh = 186d, and BBh = 187d are to be used. They are not changed from the way that they are originally initialized. The BAUDST routine then returns.

The INZCOM routine continues by sending the contents of the B register out port BAh = 186d. If the modem is specified, 2Dh = 45d is sent, and if the RS-232C connector is selected, 25h = 37d is sent. The difference is bit 3, which controls the switch selecting between these two modes of operation for the serial line.

Next the INZCOM routine sends the configuration code to the UART via port D8h = 216d. The upper three bits are masked off because they are not used.

Next the routine at 6F39h = 28,473d is called (see box). This routine clears three locations: FF40h = 65,344d, FF86h = 65,414d, and FF88h = 65,416d. The first of these locations controls the XON/XOFF protocol, the second specifies how many bytes are in the buffer for incoming characters from the serial communications line, and the third is a pointer to the position in the buffer for incoming characters.

Routine: Initialize Serial Buffer Parameters

Purpose: To initialize parameters that manage the serial communications line.

Entry Point: 6F39h = 28,473d

Input: None

Output: When the routine returns, three locations — FF40h = 65,344d, FF86h = 65,414d, and FF88h = 65,416d — are cleared.

BASIC Example:

```
CALL 28473
```

Special Comments: None

Next the INZCOM places a value of FFh = 255d in location FF43h = 65,347d and returns. This location tells the system when the serial communications line is active. The return is made by jumping to a place where there is the proper number of POPs and then a RETurn. This just happens to be the very last part of the background task discussed in Chapters 4, 5, and 6.

The function of the CLSCOM routine (see box) is the opposite of that for the INZCOM routine: CLSCOM deactivates the communications line. It is located at 6ECBh = 28,363d. It sets bits 6 and 7 of port BAh = 186d. Bit 7 hangs up the telephone, and bit 6 places the DTR (Data Terminal Ready) line of the RS-232C connector in the "not ready" state.

Routine: CLSCOM

Purpose: To deactivate the serial communications line

Entry Point: 6ECBh = 28363d

Input: None

Output: When the routine returns, the telephone connection is broken and the RS-232C DTR line is placed in the "not ready" state.

BASIC Example:

```
CALL 28363
```

Special Comments: None

Dialing the Telephone

The Model 100 dials the telephone by a method that in effect, rapidly takes the telephone on and off the "hook". A relay in the modem circuit acts as the "hook". This relay is controlled by bit 7 of port BAh = 186d. When the telephone is hung up, this bit is 1, and when the telephone is off the hook, it is 0.

After we present the ROM routines for dialing the telephone, we will provide a program that shows how you can dial the telephone from BASIC.

The routine for dialing the telephone is called DIAL and is located at 532Dh = 21,293d (see box). Upon entry the HL register pair contains the address of the telephone number. The routine first sets bit 3 of port BAh = 186d, selecting the modem instead of the RS-232C connector. Next the routine calls a routine at 5359h = 21,337d, which does the actual dialing (see box).

Routine: DIAL

Purpose: To dial the telephone

Entry Point: 532Dh = 21,293d

Input: Upon entry, the HL register points to where the telephone number is stored in memory.

Output: The routine dials the telephone number

BASIC Example:

```
CALL 21293,0,H
```

Special Comments: None

Routine: Dialing Routine

Purpose: To dial the telephone

Entry Point: 5359h = 21,337d

Input: Upon entry, the HL register pair points to where the telephone number is stored in memory.

Output: The routine dials the telephone.

BASIC Example:

```
CALL 21337,0,H
```

where H is the address of the telephone number

Special Comments: None

The dialing routine at 5359h = 21,337d calls various other routines and then gets down to the business of dialing the telephone. The dialing loop runs from 5370h = 21,360d to 539Bh = 21,403d. At the top of the loop, a check is made for the BREAK key, and the dialing routine is exited if BREAK is detected. Next, if the routine continues, a digit or other symbol is fetched from the telephone number string. It is checked to see if it is a special symbol such as CR, LF, or CTRL Z. If it is a CR/LF sequence or a CTRL Z, the routine jumps to a section of code in which the auto log on sequence is performed. If it is not, the routine skips any spaces in the telephone number and sets the carry if it finds a digit. If there was a digit, it calls a routine at 540Ah = 21,514d to dial the digit (see box).

Routine: Dial a Digit

Purpose: To dial a digit of a telephone number

Entry Point: 540Ah = 21,514d

Input: Upon entry, the digit (ASCII code or actual value) is in the A register.

Output: When the routine returns, the digit is dialed.

BASIC Example:

```
CALL 21514,A
```

where A contains the digit

Special Comments: None

The digit dialing routine at 540Ah = 21,514d prints the digit on the screen, then converts it from ASCII to a numerical value by masking off all but the lower four bits and converting zero values to ten. This numerical value is used as a loop counter to control the number of pulses that are to be sent out. Note that a zero on the telephone dial is really a ten when sent over the line.

The pulse loop begins by getting the timing delay selected by STAT (10 or 20) and using it to set a counter in the DE register pair for a pair of delay loops that control the pulse timing. If a rate of 10 pulses per second was selected, a value of 2440h = 9280d is chosen, and if a 20 pulse per second rate was selected, a value of 161Ch = 5660d is chosen. Next the pulse loop disconnects the phone by calling a routine at 52C1h = 21,185d. This is part of the "official" routine for disconnecting the phone line that begins at

52BBh = 21,179d (see box). The "disconnect" is made by setting bit 7 of port BAh = 186d equal to 1, leaving the other bits undisturbed. Next the E register of the DE pair is used to count a delay with the telephone on the hook. The pulse loop then reconnects the telephone by calling a routine at 52B4h = 21,172d, which clears bit 7 of port BAh = 186d. This is part of the "official" routine for connecting the phone line that begins at 52D0h = 21,200d (see box). Finally, the D register is used to count a delay with the telephone off the hook (connected). During the pulse loop, interrupts are disabled (using the DI instruction) because the timing loops are critical and must not be interrupted.

Routine: DISC

Purpose: To disconnect the telephone line

Entry Point: 52BBh = 21,179d

Input: None

Output: When the routine returns, the telephone line is disconnected.

BASIC Example:

```
CALL 21179
```

Special Comments: None

Routine: CONN

Purpose: To connect the telephone line

Entry Point: 52D0h = 21,200d

Input: None

Output: When the routine returns, the telephone line is connected.

BASIC Example:

```
CALL 21200
```

Special Comments: None

After the pulse loop there is a short delay to separate the digit from other digits of the telephone number. Then the dialing routine continues by checking for more special characters, including "<" for beginning of the auto log on sequence and "=" for a two-second delay.

The routine for the auto log on sequence begins at 539Eh = 21,406d (see box). It calls serial communications line routines to send and receive characters through the modem as specified by the user. The auto log on sequence then returns to the main DIAL routine.

Routine: Auto Log On

Purpose: To send auto log on sequence

Entry Point: 539Eh = 21,406d

Input: Upon entry, the HL register pair points to the descriptor for the auto log on sequence.

Output: The routine sends the auto log on sequence out the serial communications line.

BASIC Example:

```
CALL 21406,0,H
```

where H is the address where the auto log on sequence is stored in memory.

Special Comments: None

In DIAL, bit 3 of port BAh = 186d is restored to its original value, and various other business is taken care of depending upon whether you are in terminal mode or just using the computer to dial a voice call for you.

Dialing from BASIC

Here is a BASIC program that uses the DIAL routine to dial a telephone number. It prompts you for the telephone number and then dials the number for you. You can use this program as a starting point for developing more elaborate telephone routines. For example, you could write a program that waits until a specified time, dials a certain number, logs onto a computer at the other end of the line, sends some data, and then hangs up the telephone.

```
100 ' DIAL A TELEPHONE NUMBER
110 '
120   INPUT "TELEPHONE NUMBER";T$
130   S$ = T$+CHR$(13)
140   S = VARPTR(S$)
150   H = PEEK(S+1)+256*PEEK(S+2)
160   CALL 21293,0,H
170   CALL 21179
180   PRINT
190   GOTO 120
```

On line 120 of this program, the telephone number is input by the user. On lines 130-150, the address where this number is stored in memory is computed. On line 160, the DIAL routine is called with the address of this telephone number in the HL register pair. On line 170, the program hangs up the telephone, assuming that the user is on the line with a regular handset. On line 190, it loops back to try another number.

Reading from the Serial Communications Line

The serial communications line is interrupt driven. That is, whenever characters are ready to be received, the UART actuates an interrupt to a special routine to put the character in a buffer (see Figure 7-7). Whenever the computer needs a character from the communications line, it checks the buffer, not the UART directly. Let's look at the routines to handle the interrupt and to fetch characters from the buffer.

The Serial Communications Interrupt Service Routine

The interrupt service routine for the serial communications line takes in characters from the line as they are generated. Whenever the UART receives another character, it triggers an interrupt.

The Model 100 uses interrupt 6.5 for input to its serial communications line. This means that whenever the interrupt is triggered, the CPU stops what it is doing (if this interrupt is enabled) and calls location 34h = 52d. On the Model 100, the code starting at 34h = 52d disables further interrupts and jumps to location 6DACh = 28,076d.

The interrupt service routine for the serial communications line continues at location 6DACh = 28,076d (see box). It first calls a routine in RAM at F5FCh = 62,972d. This normally consists of just a RETurn instruction. Since it is in RAM, it provides a way for you to take control of the interrupt routine, perhaps placing a jump at F5FCh = 62,972d to your own interrupt routine for the serial communications line. For example, you could use such a routine in a communications program that transfers files in special formats.

Routine: Serial Interrupt Service Routine

Purpose: To read a character from the serial communications line

Entry Point: 6DACh = 28076d

Input: Upon entry, a character is ready for input from the serial communications line.

Output: When the routine returns, the character is placed in the serial communications input buffer.

BASIC Example: Not applicable

Special Comments: None

The main part of the interrupt routine manages input to a circular buffer from the serial communications line. A circular buffer is a buffer that wraps around on itself (see Figure 7-8). This particular circular buffer is 64 bytes long, starting at location FF46h = 65,350d. A pointer at FF88h = 65,416d gives the position within the buffer for bytes as they are input from the serial line. A pointer at FF87h = 65,415d gives the position

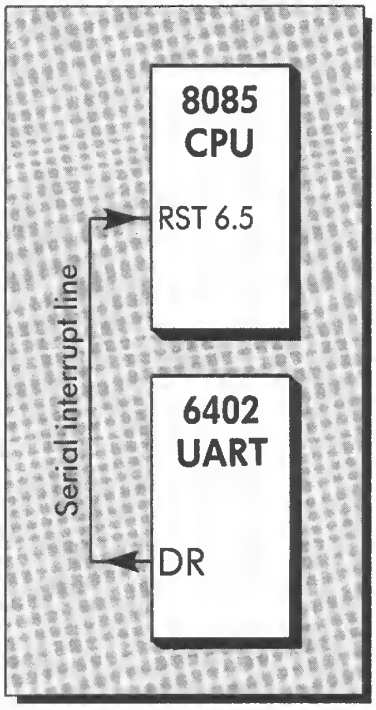


Figure 7-7. The UART interrupt

for bytes as they are removed from the buffer by other routines. Another variable, located at FF86h = 65,414d, keeps track of the number of bytes currently stored in the buffer.

The buffer management routine starts by pushing the HL, DE, BC, and PSW registers onto the stack, saving what the CPU was doing just before the interrupt occurred. Then the address 71F7h = 29,175d is pushed onto the stack, providing a return address with the proper sequence of POPs to restore the registers upon return from the interrupt. Again, this code is borrowed from the background task.

Next, port C8h = 200d is read into the accumulator. This is the data byte from the serial communications line. It will be placed in the circular buffer. First, however, it must be processed according to the choice of parity.

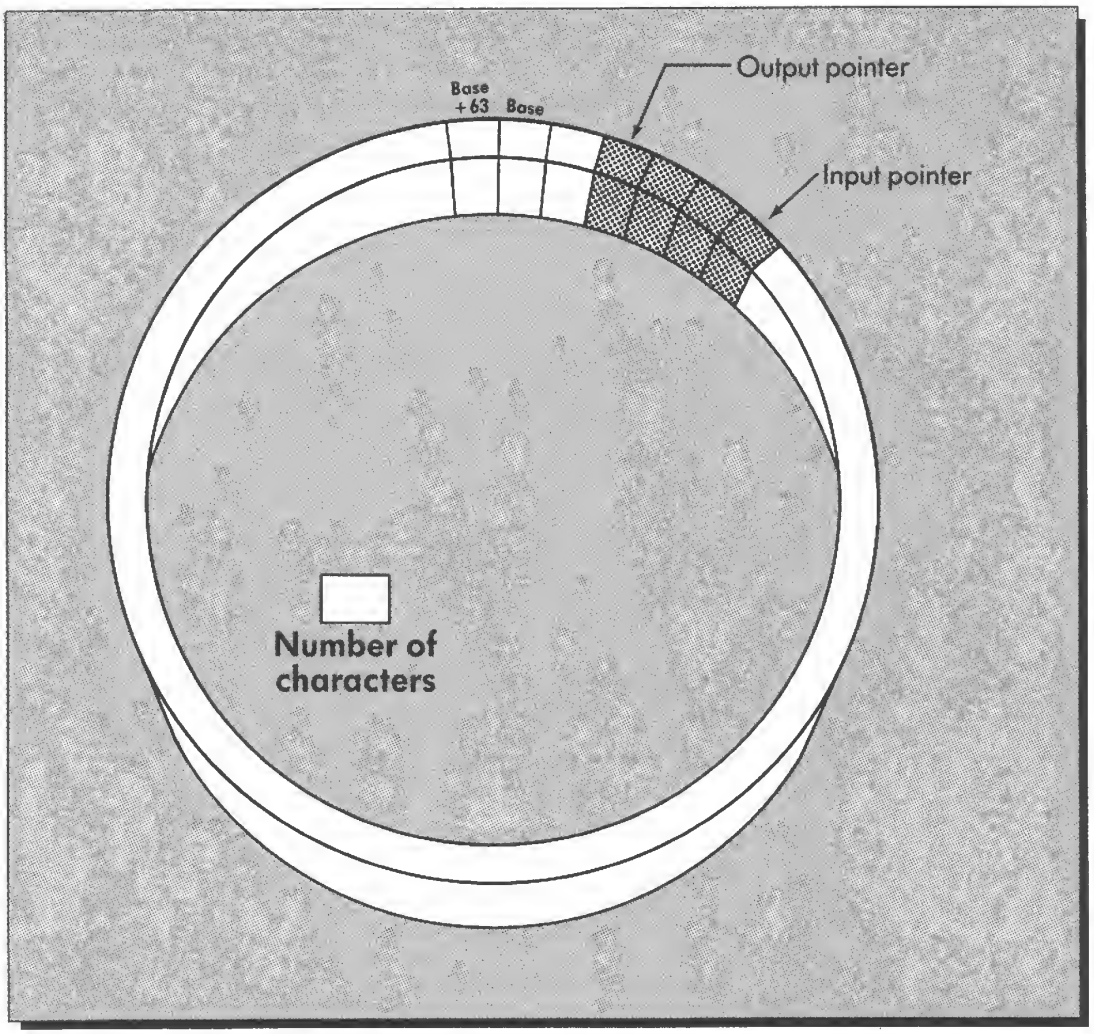


Figure 7-8. A circular buffer

The byte is ANDed with the contents of FF8Dh = 65,421d. If parity is ignored, this location contains 7Fh = 127d; otherwise, it contains FFh = 255d. In the first case it masks off the parity, and in the second case it has no effect. The result is moved to the C register.

Next, the routine checks for errors from the communications line. It reads port D8h = 216d to get the status byte of the UART. Only bits 1, 2, and 3 are used. The rest are masked off. They carry the following error signal lines: OE (Overrun Error), FE (Framing Error), and PE (Parity Error). The byte containing these bits is saved in the B register for later use.

If all of the error bits are zero, the incoming character is checked for XON/XOFF protocol characters. If the routine finds CTRL Q (XON), location FF40h = 65,344d is cleared, and if it finds CTRL S (XOFF), that location is made nonzero. Next, the routine checks location FF42h = 65,346. If this is nonzero, it returns without further action.

Next the routine checks to see if the buffer is already full. It checks location FF86h = 65,414d, which contains the number of characters currently in the buffer. If this number is equal to 64, the routine returns without further action because this indicates that the buffer is full. If the number of characters in the buffer is greater than or equal to 40, an XOFF character is sent out the communication line because the buffer is getting full. If the device at the other end of the line is listening, it should soon stop sending more characters.

Now the character is put in the buffer. First the character count (location FF86h = 65,414d) is incremented. Then a routine at 6DFCh = 28,156d is called to compute the address of the position in the buffer where the character should be put (see box), and the character is placed in the buffer at that position. The routine finishes by recording an error if the status byte sampled earlier indicates that one occurred.

The computation of the address of the position of the byte in the buffer is done in several steps. First the pointer is incremented and ANDed with 3Fh = 63d. This advances the pointer around a 64-position number wheel that uses modular or clock arithmetic. The position numbers start at 0, then 1, then 2, and so on to 63. After 63 comes 0 again. To compute the address, the base address of the buffer is added to the value of this circular pointer.

Routine: Address Computation for Circular Buffer

Purpose: To compute the address for a pointer into the circular buffer

Entry Point: 6DFCh = 28,156d

Input: Upon entry, the HL register pair points to the location in memory where the pointer is stored. The pointer is contained in a byte and has a value from 0 to 63.

Output: When the routine returns, the HL register pair contains the actual address of the buffer entry, and the DE register pair contains the address of the pointer. The value of the pointer is incremented unless it was 63, in which case it is set back to 0.

BASIC Example: Not applicable

Special Comments: None

Serial Communications Input Routines

Next, let's look at the routines that fetch characters from the buffer for use by the rest of the computer. There is a routine called RCVX to return the number of characters in the buffer and a routine called RV232C to fetch a character from the buffer. An assembly-language programmer can use these routines to send data to the serial communications line.

The RCVX routine is located at 6D6Dh = 28,013d (see box). This routine returns the number of characters currently stored in the serial communications input buffer in the A register. It also sets the Z flag accordingly. That is, if A is zero, then Z is set; otherwise, it is clear.

Routine: RCVX

Purpose: To get the number of characters currently in the serial communications input buffer

Entry Point: 6D6Dh = 28,013d

Input: None

Output: When the routine returns, the number of characters currently stored in the serial communications input buffer is in the A register, and the Z flag is set accordingly.

BASIC Example: Not applicable

Special Comments: None

The RCVX routine gets the number of characters currently in the buffer from location FF86h = 65,414d, ORing it with itself to set the zero flag if the buffer is empty (Z means empty and NZ means not empty). It also checks locations FF40h = 65,344d and FF41h = 65,345d for special conditions involving the XON/XOFF protocol.

The RV232C routine is located at 6D7Eh = 28,030d (see box). Its job is to get a character from the serial communications input buffer. Upon exit the A register contains the ASCII code of the character from the buffer. The zero flag is set (Z) if there is no error and clear (NZ) if there was an error. The carry is set (C) if the BREAK key was hit and clear (NC) otherwise.

Routine: RV232C

Purpose: To get a character from the serial communications input buffer

Entry Point: 6D7Eh = 28,030d

Input: None

Output: When the routine returns, the A register contains the ASCII code of the character from the buffer. The zero flag is set (Z) if there is no error and clear (NZ) if there was an error. The carry is set (C) if the BREAK key was hit and clear (NC) otherwise.

BASIC Example: Not applicable

Special Comments: None

The RV232C routine pushes the HL, DE, and BE registers on the stack and pushes the address 71F8h = 29,176d on the stack for the return address. This is where the proper number of POPs and a RETurn are located.

Next the RV232C routine goes into a loop in which it waits for a character from the buffer. The loop first calls a routine at 729Fh = 29,343d to check for a BREAK key, returning with the carry set if it detects this key (see box). If there was no BREAK, the RCVX routine is called to check the queue. If the queue is empty, the loop keeps looping. The loop will exit normally as soon as there is a character in the buffer.

Routine: BREAK check

Purpose: To check for BREAK key

Entry Point: 729Fh = 29,343d

Input: None

Output: The routine returns with the carry flag set if it detects the BREAK key.

BASIC Example: Not applicable

Special Comments: None

If there are fewer than three characters in the buffer, a routine called SEND Q, at 6E0Bh = 28,171d, is called to send CTRL Q (XON) out the communications line (see box later in the chapter).

The RV232C then gets the next character out of the buffer, using the routine at 6DFCh = 28,156d (described earlier) to compute its address within the buffer. It checks for an error condition left by the interrupt service routine and then returns.

Writing to the Serial Communications Line

There are two routines for sending characters out the serial communications line: SNDCOM and SD232C. The first simply sends characters, while the second uses the XON/XOFF protocol. If you are an assembly-language programmer, you can use these routines to send bytes out the serial communications line.

The SNDCOM routine is located at 6E3Ah = 28,218d (see box). It sends a single byte out the serial communications line, either to the modem or to the RS-232C connector, whichever is currently selected. It expects the character in the C register.

Routine: SNDCOM

Purpose: To send a character out the serial communications line

Entry Point: 6E3Ah = 28,218d

Input: Upon entry, the ASCII code of the character is in the C register.

Output: The routine sends the character out the serial communications line.

BASIC Example: Not applicable

Special Comments: None

The SNDCOM routine contains a loop that waits for the communications line to be ready to output the next character. This loop calls the BREAK detector routine at 729Fh = 29,343d (described previously) and then reads port D8h = 216d, checking bit 4 to see if the UART is free to accept the next character. If the bit is zero, the loop continues looping; otherwise, it does a normal exit, and the routine continues and sends the character out port C8h = 200d before it returns.

The SD232C routine is located at 6E32h = 28,210d (see box). It sends a character out the communications line using the XON/XOFF protocol.

Routine: SD232C

Purpose: To send a character out the communications line using the XON/XOFF protocol

Entry Point: 6E32h = 28,210d

Input: Upon entry, the ASCII code of the character is in the A register.

Output: The character is sent out the serial communications line.

BASIC Example:

```
CALL 28210,A
```

where A is the ASCII code of the character.

Special Comments: None

The SD232C routine calls a routine at 6E4Dh = 28,237d, which does the protocol. This routine waits if the communications line is to be held up by an XOFF. If a BREAK is detected, it returns with the carry flag set. In this case, the SD232C routine exits without sending the character; otherwise, it runs on into the SNDCOM routine to send the character.

Serial Transmission from BASIC

Here is a BASIC program that sends characters out the serial communications line. It prompts the user for a string, sends it out the serial communications line, and then loops back for another string.

```
100 ' SEND BYTES TO SERIAL PORT
110 '
120   PRINT "STRING TO SEND"
130   INPUT T$
140   T = VARPTR(T$)
150   X0 = PEEK(T)
160   X1 = PEEK(T+1)+256*PEEK(T+2)
170   FOR X = X1 TO X1+X0-1
180    CALL 28210,PEEK(X)
190   NEXT X
200   GOTO 120
```

On lines 120-130 of this program, the string is input. On lines 140-160, the length and address of the string are computed. The variable X0 contains the length of the string, and the variable X1 contains its address. On lines 170-190, the bytes of the string are sent by getting them one by one from memory and calling the SD232C routine each time. On line 200, the program loops back for the next string.

BASIC also has ways of sending bytes to the communications line without directly calling machine language. The following program shows one such method.

```
100 ' SEND BYTES TO COM LINE
110 '
120   OPEN "COM:68N2E" FOR OUTPUT AS #1
130   PRINT "STRING TO SEND"
140   INPUT T$
150   PRINT #1, T$;
160   GOTO 130
```

This program is shorter and easier to follow than the previous one, but of course it does not reveal much about the inner workings of the ROM.

Protocal Routines

Now let's look at the main protocol routine. This routine first checks location FF42h = 65,346d. If this location is zero, then the communications line is in a "go" condition, and the protocol routine returns ready to let the output of the character occur. If this location is nonzero, the line is in the "stop" condition. In that case, the character is checked to see if it is a CTRL Q (XON). If it is, the routine zeros locations FF8Ah = 65,418d and FF41h = 65,345d and returns. If not, it checks for CTRL S (XOFF). If the character is found, a value of FFh = 255d is stored in location FF41h = 65,345d, and the routine returns. If not, it goes into a loop that waits for either the BREAK key to be detected (returning with the carry set) or location FF40h = 65,344d to be zero.

Let's finish with two other protocol routines: SENDCQ and SENDCS. The first sends an XON (CTRL Q), and the second sends an XOFF (CTRL S) to the serial communications line.

The SENDCQ routine is located at 6E0Bh = 28,171d (see box). It first checks location FF42h = 65,346d. If this is zero, it returns. If not, it checks location FF8Ah = 65,418d. If this is not one, it returns without further action. If this location is one, the routine stores a zero in FF8Ah = 65,418d and returns, sending out an XON character.

Routine: SENDCQ

Purpose: To turn on the XON/XOFF protocol for incoming characters from the serial communications line

Entry Point: 6E0Bh = 28,171d

Input: None, except for the XON/XOFF control variables FF42h = 65,346d and FF8Ah = 65,418d

Output: The routine turns on the XON/XOFF protocol, sending an XON character (ASCII 11h = 17d) out the serial communications line if needed.

BASIC Example:

```
CALL 28171
```

Special Comments: None

The SENDCS routine is located at 6E1Eh = 28,190d (see box). Like SENDCQ, it first checks location FF42h = 65,346d. Again, if this is zero, it returns. Otherwise, it checks location FF8Ah = 65,418d. This time it returns if this location is not zero. If it is zero, it sets this location to one and returns, sending out an XOFF character.

Routine: SENDCS

Purpose: To turn off the XON/XOFF protocol for incoming characters from the serial communications line

Entry Point: 6E1Eh = 28,190d

Input: None, except for the XON/XOFF control variables FF42h = 65,346d and FF8Ah = 65,418d

Output: The routine turns off the XON/XOFF protocol, sending an XOFF character (ASCII 13h = 19d) out the serial communications line if needed.

BASIC Example:

```
CALL 28190
```

Special Comments: None

Summary

In this chapter we have studied the two serial communications channels for the Model 100 computer: the modem, which connects the Model 100 to the telephone, and the RS-232C connector, which provides a standard method for connecting the Model 100 directly to other computers.

We have seen that both communications channels are handled by the same UART (Universal Asynchronous Receiver Transmitter) chip. We have shown how to switch between the two channels, how to initialize the UART, and how to send and receive characters through it. We have also studied the circular buffer that manages the flow for incoming characters and the XON/OFF protocol that is used to prevent the buffer from overflowing.

8

Hidden Powers of Sound

Concepts
How sound works in the Model 100

ROM Routines for Sound
The BEEP command
The SOUND command

Sound makes computers more friendly. On the simplest level, for instance, a "beep" sound can inform you that you've made an error while entering data. In games, sound can provide reinforcement for winning plays. Sound is likely to be an integral part of the input/output systems for personal computers in the future.

In this chapter we will explore the secrets of sound on the Model 100. You will see how to turn the sound on and off through bits on ports of the 8155 PIO and how to program the timer that produces a tone for the sound circuit. These techniques have the potential of providing much more versatile control of sound than can be obtained using BASIC. We will also look at how the BEEP and SOUND commands work.

How Sound Works in the Model 100

The sound circuits in the Model 100 consist of a timer and two switches (see Figure 8-1). The timer is the same one that generates the baud rate for the UART, as explained in the previous chapter, and it is programmed in the same way.

The first of the two switches is controlled by bit 5 of port BAh = 186d. It turns the sound on and off. A value of 1 turns it on and a value of 0 turns it off.

The second switch is controlled by bit 2 of port BAh = 186d. It connects and disconnects the output of the timer from the sound circuit. A value of 0 makes the connection and a value of 1 breaks the connection.

The following BASIC program demonstrates how the switches and the timer can be programmed directly to make sounds of various frequencies. It asks you for the frequency divisor D, which is the value loaded into the timer. The actual frequency is given by the formula:

2,457,600/D

where D is the divisor that you specify.

```
100 ' MAKE A SOUND
110 '
120   INPUT "FREQUENCY DIVISOR";D
130 '
140 ' PROGRAM THE TIMER
150   OUT 188,(D MOD 256)
160   OUT 189,((D/256) AND 127) OR 64
170   OUT 184,195
180 '
```

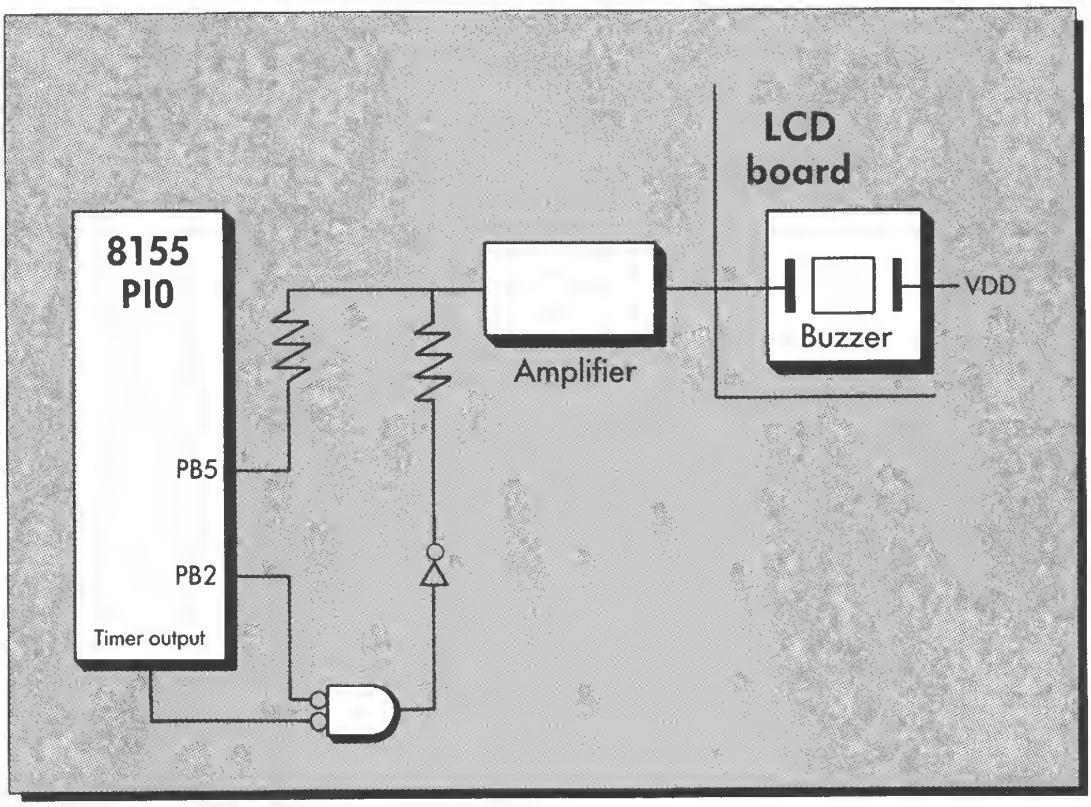


Figure 8-1. The sound circuit of the Model 100

```
190 ' TURN THE SOUND ON
200   X = INP(186)
210   OUT 186, (X AND 219) OR 32
220   GOTO 120
```

Let's look at this program in more detail. The frequency divisor D is input on line 120. On line 150, the lower eight bits are sent to port BCh = 188d, and on line 160, the upper six bits are sent to port BDh = 189d. In addition, bits 6 and 7 of port BDh = 189d are set to binary 01. This ensures that a square wave is produced.

In line 170, the timer is started by sending C3h = 195d to port B8h = 184d. This port programs the way the timer and the ports on the 8155 PIO chip will be used. The upper two bits program the timer, and the lower six bits program the ports. The pattern C3h = 195d ensures that the parallel ports of the PIO will remain as they are.

In lines 200-210, bit 5 is set to one and bit 2 is cleared in port BAh = 186d. This makes the connections from the timer to the sound circuit.

In line 220, the program loops back to get the next tone.

The ROM Routines for Sound

ROM routines for sound are used to implement the BEEP and SOUND commands in BASIC.

The BEEP Command

Let's start with the "beep" sound. This can be actuated either by the BEEP command or by sending a CTRL G (BEL) to the screen printing routines.

The code for the BEEP command starts at 4229h = 16,937d (see box). It simply sends an ASCII 7 (BEL) character to the screen printing routines via the RST 4 command (see Chapter 4). The screen printing routines dispatch to the routines for control characters in the code from 4373h = 17,267d to 4389h = 17,289d, using a table that starts at 438Ah = 17,290d. The routine for BEL is located at 7662h = 30,306d.

Routine: BEEP

Purpose: To sound a beep

Entry Point: 4229h = 16,937d

Input: None

Output: To the sound system

BASIC Example:

```
CALL 16937
```

Special Comments: None

The physical routine to generate the Beep is at 7662h = 30,306d. Here the routine at 765Ch = 30,300d is called to turn off interrupts (see box). Then the B register is cleared to set up a loop count of 256 for the BEEP loop, which is next. This loop first calls a routine at 7676h = 30,326d to flip bit 5 of port BAh = 186d (see box). Then it calls a routine at 7657h = 30,295d (see box) to produce a short delay (the C register is loaded with a timing count of 80). The loop executes 256 times and then returns, turning on the interrupts.

Routine: Flip the Sound Bit

Purpose: To change the sound switch

Entry Point: 7676h = 30,326d

Input: None

Output: To the sound system

BASIC Example:

```
CALL 30326
```

Special Comments: None

Routine: Sound Delay

Purpose: To produce a delay

Entry Point: 7657h = 30,295d

Input: Upon entry, the C register contains a delay count.

Output: The routine returns about 5.7*C + 10.2 microseconds after it has been called, causing a delay of that length.

BASIC Example: Not applicable

Special Comments: None

Note that this method of producing a tone does not use the timer.

Bit flipping, which rapidly opens and closes the sound circuit, provides an alternate method of generating sounds. Here is a BASIC program that illustrates this bit-flipping method. The result sounds almost like a machine gun. The slightly irregular pattern occurs because the background task has not been turned off. You can gain better control of the sound system if you write in machine language. Starting with the ideas in this BASIC program, you can develop machine-language programs to produce much more interesting and sophisticated sounds.

```
100 ' FLIP THE BITS
110 '
120   FOR I = 0 TO 20
130    FOR J=1 TO 50:CALL 30326:NEXT
140    FOR J=1 TO 10:NEXT
150   NEXT I
```

The program consists of a nested loop structure. The large loop extends over lines 120-150. This loop produces twenty pulses of sound. Each pulse is generated by line 130, which calls the bit-flipping routine fifty times. A short pause between the pulses is generated by line 140.

The SOUND Command

Now let's look at the SOUND command. The code for this command begins at 1DC5h = 7621d (see box). Here the commands SOUND ON and SOUND OFF are checked for. If neither is indicated, the frequency and length parameters are loaded, and the routine jumps to location 72C5h = 29,381d (see box). At this location you can find a routine called MUSIC.

Routine: SOUND

Purpose: To control the sound or make a note (depending upon the syntax)

Entry Point: 1DC5h = 7621d

Input: Upon entry, the HL register pair points to the end of the tokenized SOUND command line (right after the token for SOUND).

Output: Depending upon the syntax of the command line, one of the following BASIC commands is executed: SOUND, SOUND ON, or SOUND OFF.

BASIC Example:

```
CALL 29381,0,H
```

where H is the address of the end of the tokenized command line for a SOUND command.

Special Comments: None

Routine: MUSIC

Purpose: To play a note of given frequency and duration

Entry Point: 72C5h = 29,381d

Input: Upon entry, the DE register pair contains a pitch number as described on page 180 of the Model 100 owner's manual, and the B register contains the duration in approximately 1/50ths of a second.

Output: To the sound system

BASIC Example: Not applicable

Special Comments: None

The MUSIC routine expects the frequency divider in the DE register pair and the duration in the B register. The routine first disables interrupts with the DI instruction. This is needed because of the timing loop. The contents of the DE register pair are then loaded into ports BCh = 188d and BDh = 189d to set the timer, with the E register going to BCh = 188d and the D register (ORed with 40h = 64d) going to BDh = 189d. This is much

like lines 150-160 of our "Make a Sound" BASIC program. Also as in our BASIC program, the value C3h = 195d is sent to port B8h = 184d to start the timer. Next, bit 5 is set and bit 2 is cleared in port BAh = 186d, again as in the BASIC program. A break check is made by calling the routine at 729Fh = 29,343d (see box). Then a timing loop at 72EAh = 29,418d counts the specified delay. Finally, the tone is turned off by setting bit 2 of port BAh = 186d and resetting the baud rate into the timer.

Summary

In this chapter we have shown how to program the sound circuits of the Model 100, including two BASIC example programs. We have also shown how the BEEP and SOUND commands are implemented in ROM routines.

9

Hidden Powers of the Cassette

Concepts

How the cassette interface works
The SIM instruction and the SOD line
The RIM instruction and the SID line

ROM Routines for the Cassette System

Motor control
Read routines
Write routines

The tape cassette interface provides an inexpensive way to save and retrieve programs and other kinds of files on the Model 100. In this chapter we will explore the powers of the cassette interface. We will show how to turn the cassette motor on and off through bits on ports of the 8155 PIO chip. We will also explain how to read and write data to the cassette.

How the Cassette Interface Works

The hardware that interfaces the Model 100 to a tape cassette player has three major components: motor control, writing data, and reading data.

The motor control circuit consists of a relay driven by an amplifying transistor, which in turn is controlled by bit 3 of port E8h = 232d (see Figure 9-1). A value of 0 in this bit turns off the cassette motor, and a value of 1 turns it on.

The circuit for writing data to the cassette player converts two-level digital information from the SOD (Serial Out Data) pin on the 8085 CPU

to a waveform suitable for recording on cassette tape (see Figure 9-2). This signal is sent out the cassette connector to the AUX input of the cassette recorder.

The SOD pin is activated by the SIM instruction of the 8085 CPU. SIM is a dual-purpose instruction. In addition to controlling the SOD pin, it is used to set interrupts 7.5, 6.5, and 5.5 (see Chapter 3). Before the SIM instruction is used, the accumulator must be loaded with a bit pattern. If bit 3 is set, the instruction is used to control interrupts; if bit 6 is set, it is used to control the SOD line, sending the value of bit 7 there.

The circuit for reading data from the cassette player converts the signal from the cassette player back into two-level digital information for input into the SID (Serial In Data) pin of the 8085 CPU (see Figure 9-2).

The SID pin is monitored by the CPU's RIM instruction. After this instruction is executed, the value of the SID line is found in bit 7 of the accumulator.

In the next section we will explore the routines that the Model 100 uses to write and read data bytes serially through these circuits.

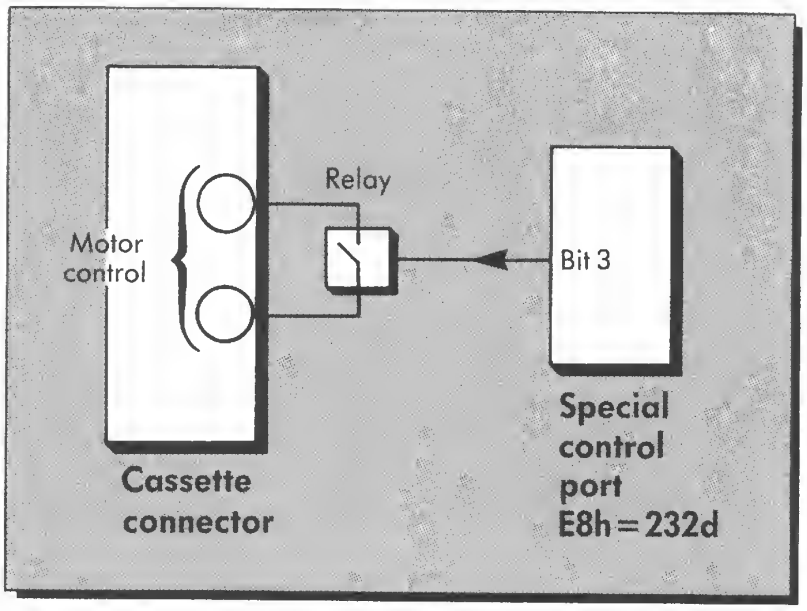


Figure 9-1. Motor control circuit

The ROM Routines for the Cassette System

Let's start with motor control. Then we'll move on to the read and write routines.

Turning the Cassette Motor On and Off

The cassette motor circuit is normally used to do what its name implies, namely, to turn the cassette motor on and off while reading or writing data

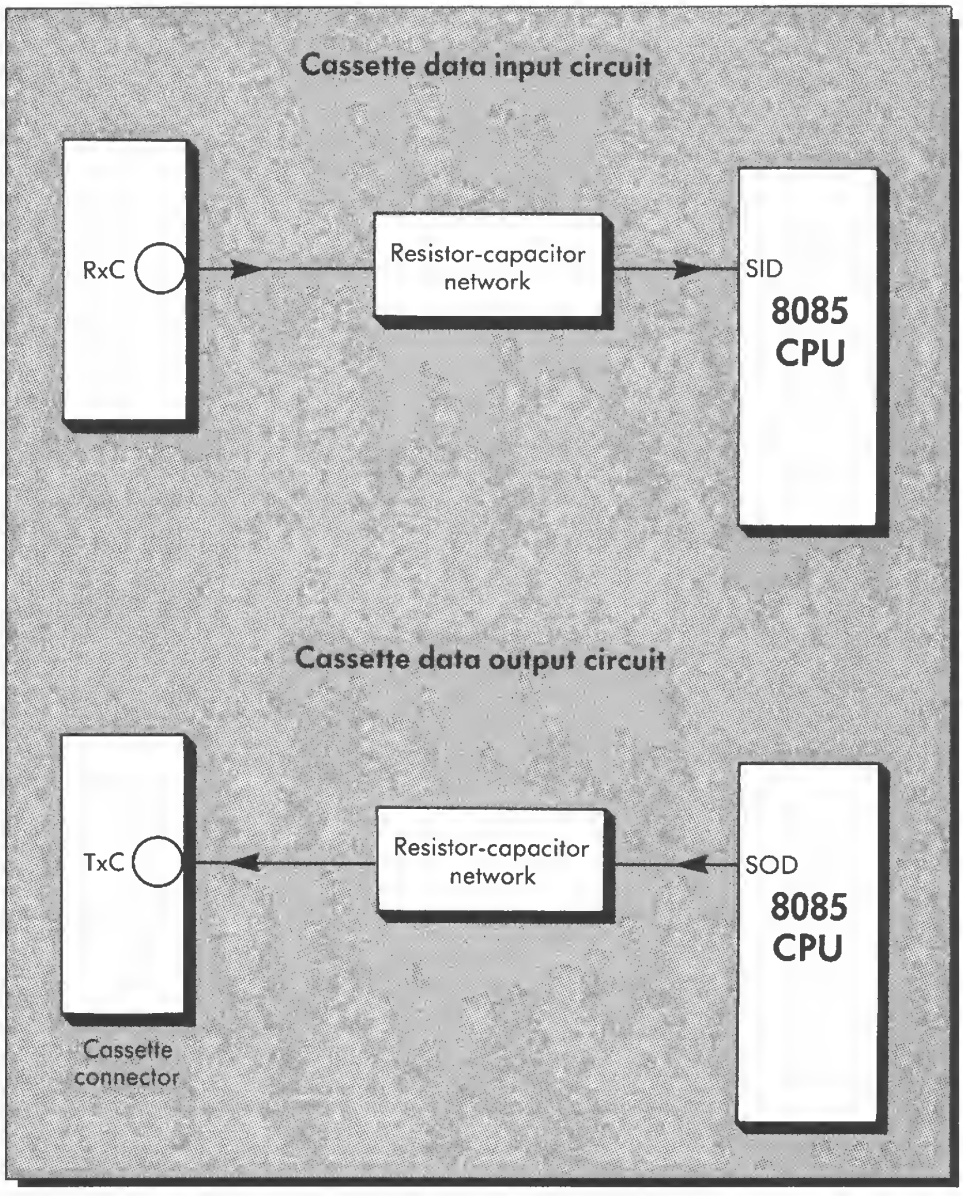


Figure 9-2. Cassette data output and input circuits

to the cassette player/recorder. However, this motor circuit can be used to control other devices as well, such as relays that turn on and off appliances or lab equipment.

The routine to turn on the tape cassette motor is called CTON and is located at 14A8h = 5288d (see box). This routine disables interrupts using the DI instruction, loads a nonzero value into the E register, and then jumps to a routine called REMOTE, which does the action.

Routine: CTON

Purpose: To turn on the tape cassette motor

Entry Point: 14A8h = 5288d

Input: None

Output: The tape cassette motor circuit is turned on.

BASIC Example:

```
CALL 5288
```

Special Comments: None

The REMOTE routine is located at 7043h = 28,739d (see box). It turns the cassette motor on or off. Upon entry, if the E register is nonzero, it turns on the cassette motor. If the E register is zero, it turns off the cassette motor.

Routine: REMOTE

Purpose: To control the tape cassette motor

Entry Point: 7043h = 28,739d

Input: Upon entry, the E register indicates whether the motor is to be turned on or off. A zero value in E indicates off, and a nonzero value in E indicates on.

Output: To the tape cassette motor circuit

BASIC Example: Not applicable

Special Comments: None

The REMOTE routine uses bit 3 of port E8h = 232d to control the cassette motor. Other bits in this port control other devices such as the optional ROM (bit 0), the printer (bit 1), and the clock (bit 2). To make sure that the REMOTE routine changes only this one bit, a copy of the contents of port E8h = 232d is maintained in memory at location FF45h = 65,349d. The REMOTE routine first reads the contents of this location into the accumulator and then clears bit 3 of the accumulator. Next, it checks the E register. If register E is nonzero (as it is coming from the CTON routine), the routine ORs the accumulator with 8, setting bit 3. If register E is zero, it leaves the accumulator as it was, with bit 3 clear. In either case, it sends the result to port E8h = 232d and also stores it in location FF45h = 65,349d before returning.

The routine to turn the tape cassette off is called CTOFF and starts at location 14AAh = 5290d (see box). This entry point is right in the middle of a CPU instruction that belongs to the CTON command, so the code differs, even though it is shared by the two routines. This time it enables interrupts using the EI instruction and then loads zero into the E register before jumping to the REMOTE routine. This time the REMOTE routine clears bit 3 of port E8h = 232d, turning off the cassette motor.

Routine: CTOFF

Purpose: To turn the tape cassette motor off

Entry Point: 14AAh = 5290d

Input: None

Output: The tape cassette motor circuit is turned off.

BASIC Example:

```
CALL 5290
```

Special Comments: None

Here is a BASIC program that directly controls bit 3 of memory location FF45h = 65,349d, turning it on and off. BASIC sends this byte out port E8h = 232d as part of its normal housekeeping; thus we do not have to explicitly do so ourselves in this program. You should note that BASIC has built-in commands to turn the cassette motor on and off; so it is not normally necessary to get down to this level.

```
100  ' CASSETTE MOTOR CONTROL
110  '
120    INPUT "BIT VALUE FOR MOTOR";X
130    Y = PEEK(65349)
140    Y = (Y AND 247) OR 8*(X AND 1)
150    POKE 65349,Y
160    GOTO 120
```

On line 120 of this program, we get the bit value (0 or 1) for the motor control bit. On line 130, we get the contents of memory location FF45h = 65,349d. This is the Model 100's record of port E8h = 232d. On line 140, we insert the new bit value in the byte, and on line 150, we put it back into location FF45h = 65,349d. On line 160, we loop back for another bit value.

Writing to the Cassette

The routines to write to the cassette are at several levels and involve two kinds of activity: 1) sending ordinary data bytes and 2) sending special header bytes.

Ordinary Data

Let's start with the first case, sending ordinary data bytes to the cassette recorder. The lowest-level routine for doing this is called DATAW and is located at 6F5Bh = 28,507d (see box).

Routine: DATAW

Purpose: To send a byte to the cassette recorder

Entry Point: 6F5Bh = 28,507d

Input: Upon entry, the A register contains the byte to be written.

Output: The byte is sent to the tape cassette recorder.

BASIC Example:

```
CALL 28507,A
```

where A contains the data byte that is to be sent to the tape cassette recorder.

Special Comments: None

The DATAW routine expects the outbound byte in the A register. If the BREAK key was hit during the routine, it returns with the carry set; otherwise it returns with the data byte sent and the carry flag clear.

For each outbound byte, the DATAW routine sends nine cycles of an electrical signal to the cassette recorder. The first cycle is a synchronizing signal corresponding to a bit value of zero, and each of the remaining eight cycles corresponds to one of the eight bits in the outbound byte. For each bit, a value of zero is sent as a single cycle that lasts about 837 microseconds, and a value of one is sent as a single cycle that lasts about 418 microseconds. Notice that the cycle for a value of 1 is just about one half the length of the cycle for the value 0. The corresponding frequencies are about 1,195 cycles per second for 0 and about 2,391 cycles per second for 1.

The individual bits are sent via a routine located at 6F6Ah = 28,522d (see box). This routine rolls the accumulator left one position, bringing what was the leftmost bit into the carry. The carry then contains the next bit to be sent. If the carry is clear, a value of 4349h = 17,225d is loaded into the DE register pair; otherwise, a value of 1F24h = 7972d is left in DE. These values control timing loops governing how long the SOD line is kept low and how long it is kept high. In other words, these values control the shape of the waveform that is sent through the SOD line.

Routine: Write Cassette Data Bit

Purpose: To send an individual bit to the tape cassette recorder

Entry Point: 6F6Ah = 28,522d

Input: Upon entry, bit 0 of the A register contains the bit to be sent to the tape cassette recorder.

Output: The bit is sent to the tape cassette recorder.

BASIC Example:

```
CALL 28522,A
```

where A contains the value of the A register.

Special Comments: None

Let's look at this routine in more detail. During the first timing loop (controlled by the D register), the SOD signal is assumed to be low. Right after this loop, the SIM instruction is used to raise the SOD line to a high value of 1. Next, the second timing loop (controlled by the E register) delays

while the SOD line retains the value 1. Then the SIM instruction is used to bring the SOD line low again. Finally, the routine jumps to the BREAK check routine at 729Fh = 29,343d (see Chapter 6) for its return.

The timing values put into the D and E registers have been carefully chosen to account for delays caused not only by the timing loops themselves but by the surrounding code. We found that for a bit value of 0, the SOD line was held low for 1027 CPU clock cycles and high for 1030 CPU clock cycles. For a bit value of 1, the SOD line was held low for 516 cycles and high for 512 cycles. Although the high/low times vary somewhat irregularly, the total number of cycles for a bit value of 0 is 2057, which is just about twice the total of 1028 cycles that we found for a bit value of 1. Since the CPU's clock runs at 2,457,600 cycles per second, you can compute how long each cycle is. As mentioned before, the cycle times are about 837 and 418 microseconds, respectively.

The CSOUT routine is the next higher level routine for sending data bytes to the cassette recorder. It is located at 14C1h = 5313d (see box). In addition to sending out bytes to the cassette, it manages a special error checking byte called a *checksum*.

Routine: CSOUT

Purpose: To send a data byte to the tape cassette recorder and update the checksum

Entry Point: 14C1h = 5313d

Input: Upon entry, the data byte is in the A register, and the current checksum is in the C register.

Output: The data byte is sent to the tape cassette recorder and the checksum is updated.

BASIC Example: Not applicable

Special Comments: None

Let's see how the checksum works. Bytes are sent to the cassette in blocks. For a BASIC program saved in regular non-ASCII form, the whole program is sent as one block. For ASCII files, however, the bytes are packaged in 256-byte blocks. The checksum is computed for each block as the lower eight bits of the sum of all the bytes in the block. This is not as complicated as it sounds, because the sum is computed using an eight-bit register; thus, only the lower eight bits of the answer are retained.

The negative of the checksum byte is placed at the end of the block. When the block is read, all the bytes of the block, including the byte from the checksum, are summed into a byte. The result should be zero; if it is not, an error must have occurred. If the result is zero, there is only a small chance that there is an error.

The CSOUT routine expects the outbound byte in the A register and the previous value of the checksum in the C register. It returns with the updated checksum in the C register.

The CSOUT routine pushes both the DE and HL register pairs on the stack. It then adds the value of the outbound byte to the the previous value of the checksum in the C register, and the result is placed back in the C register. Next, the DATAW routine is called to put the byte out to the cassette. If upon return from this routine the carry is set, the cassette motor is turned off by calling CTOFF and an IO error is declared. This happens only if a BREAK is detected. If the carry is clear, the CSOUT routine returns, POPping register pairs so that the BC, DE, and HL registers are all preserved.

It is technically possible to write BASIC programs that send bytes to the cassette recorder by calling the DATAW routine. However, BASIC is too slow to attain the close timing of bytes that would be required for recording actual data.

Synchronizing Header

Since bits are recorded serially on tape, there has to be a way for the cassette-reading logic to find and lock onto the first and then the subsequent bits of each byte. Some computers use UARTs and modems to create synchronizing information for recording each byte serially on tape. However, the Model 100 uses much simpler hardware and synchronizes entire blocks of bytes.

Each block of data begins with a series of special bit patterns that align the timing of the Model 100's cassette tape reading software so that it hits the right place in the code to accept the first bit of the first byte of each block of data. These synchronizing bit patterns make up the header bytes.

Let's look at the routine to write the special header bytes. This routine is called SYNCW, and it is located at 6F46h = 28,486d (see box). It consists of a loop that puts out 512 bytes with the value 55h = 85d and one byte with the value 7Fh = 127d. It calls a routine at 6F5Eh = 28,510d to send out the bytes (see box). This routine is essentially the DATAW routine without the first synch bit. The effect is a rapidly alternating pattern of zero and one bit values followed by a byte value of 7Fh = 127d.

Routine: SYNCW

Purpose: To write the synchronizing header to the tape cassette recorder

Entry Point: 6F46h = 28,486d

Input: None

Output: The synchronizing header is written to the tape cassette recorder.

BASIC Example:

```
CALL 28486
```

Special Comments: None

The actions of turning on the cassette motor and sending the special header are combined in one routine located at 148Ah = 5258d (see box). This routine first calls the CTON routine and then calls the SYNCW routine. If the SYNCW routine returns with an error (carry set because of a BREAK during DATAW), it calls the error routine at 45Dh = 1117d with the code for "IO error" in the E register. If there was no error, the routine simply returns.

Routine: Start Cassette Write

Purpose: To start writing to the tape cassette recorder

Entry Point: 148Ah = 5258d

Input: None

Output: The tape cassette motor is turned on, and the synchronizing header is written to the tape cassette recorder.

BASIC Example:

```
CALL 5258
```

Special Comments: None

Reading from the Cassette

The routines to read data from the cassette, like those for writing, consist of those for ordinary data bytes and those for the synchronizing header.

Ordinary Data

Let's start with the ordinary data bytes. The DATAR routine, with entry point at 702Ah = 28,714d, is at the lowest level (see box). It returns the incoming byte in the D register. If the BREAK key was hit, DATAR returns with the carry set; otherwise, it returns with the carry clear.

Routine: DATAR

Purpose: To read a byte from the tape cassette player

Entry Point: 702Ah = 28,714d

Input: None

Output: When the routine returns, the D register contains the data byte from the tape cassette player.

BASIC Example: Not applicable

Special Comments: None

The DATAR routine consists of two loops. The first loop waits for the synch bit, and the second loop picks up the eight data bits.

The DATAR routine calls a routine at 6FDBh = 28,635d to pick up the individual bits from the cassette player (see box). This bit-reading routine returns a count in the C register that measures the length of an incoming square-wave pulse. A count of 21 or more indicates a 0 bit value, and a count of less than 21 indicates a 1 bit value.

Routine: Read Cassette Data Bit

Purpose: To read a bit from the tape cassette player

Entry Point: 6FDBh = 28,635d

Input: None

Output: When the routine returns, the C register contains a number that measures the length of an incoming square-wave pulse. A count of 21 or more indicates a 0 bit value, and a count of less than 21 indicates a 1 bit value.

BASIC Example: Not applicable

Special Comments: None

The bit-reading routine has two major parts: one part counts pulses that go low-high-low, and the other part counts pulses that go high-low-high. The DATAR routine uses only the first part. In this part, there are two loops. The first loop waits for a high on the SID line, and the second loop measures how long the SID line stays high.

The first loop of the bit-reading routine looks for a BREAK by calling the BREAK check routine at 729Fh = 29,343d and monitors the SID line by executing the RIM instruction. The RIM instruction leaves the value of the SID signal line in bit 7 of the accumulator. As soon as the SID line goes high, the routine leaves its first loop and starts the second loop.

The second loop of the bit-reading routine counts the number of times that it loops while the SID line is still high. Each execution of the loop takes about 29 CPU clock cycles; thus, a count of about 17 corresponds to a 1 bit, and a count of about 35 corresponds to a 0 bit. If the count gets as high as 256, it starts all over again with the first loop. If not, it returns with the count, calling a routine at 7676h = 30,326d to flip the sound bit to make a click for you to hear (see Chapter 8). If the SOUND OFF command is currently in force, it skips this bit flip. Location FF44h = 65,348d is used to control the SOUND ON/OFF feature during cassette operation.

After the bit-reading routine, the DATAR routine calls a routine at 7023h = 28,707d to check the count and pack the bits, one at a time, into the D register (see box). This routine checks the count in the C register, using the CPI instruction to compare the count against the value of 21. This places the correct bit value into the carry. It then rotates this bit value from the carry into the D register via the accumulator.

Routine: Pack Cassette Data Bit

Purpose: To pack serial bits from the cassette into the D register

Entry Point: 7023h = 28,707d

Input: Upon entry, the C register contains the count from the bit-reading routine, and the D register contains the partially packed data byte.

Output: When the routine returns, the bit is placed into bit 0 of the data byte, and the previous contents are shifted left by one position.

BASIC Example: Not applicable

Special Comments: None

The CASIN routine at 14B0h = 5296d is the next higher level routine for reading data bytes from the cassette recorder (see box). As it reads data bytes from the cassette, it computes the checksum byte.

Routine: CASIN

Purpose: To read a data byte from the tape cassette player and update the checksum

Entry Point: 14B0h = 5296d

Input: Upon entry, the C register contains the current checksum.

Output: When the routine returns, the data byte is in the A register, and the updated checksum is in the C register.

BASIC Example: Not applicable

Special Comments: None

Upon entry, the CASIN routine expects the current value of the checksum byte in the C register. It returns with the updated checksum in the C register and the data byte in the A register.

The CASIN routine saves the DE, HL, and BC register pairs on the stack. It calls the DATAR routine to read the data byte. If a BREAK is detected, DATAR returns with the carry set, and CASIN jumps to declare an IO error. If there was no BREAK, CASIN continues, adding the value of the data byte to the current checksum and placing the result back into C.

Synchronizing Header

The routine to read the special synchronizing header is called SYNCR and is located at 6F85h = 28,549d (see box). It is designed to wait for this special header signal.

Routine: SYNCR

Purpose: To read the synchronizing header from the tape cassette player

Entry Point: 6F85h = 28,549d

Input: From the cassette player

Output: None

BASIC Example:

```
CALL 28549
```

Special Comments: None

The SYNCR routine is more complicated and tricky than the corresponding SYNCW routine, which was used to generate the header. The SYNCR routine consists of three loops. The first two lock into the alternating pattern of 0 and 1 bits in the body of the header, and the last loop checks for the last byte of the header, which is 7Fh = 127d.

The actions of turning on the cassette motor and detecting the special header are combined in one routine, located at 1499h = 5273d (see box). This routine calls the CTON routine, waits for almost a second to let the tape get up to speed, and then jumps to the SYNCR routine to look for the header.

Routine: Start Cassette Read

Purpose: To turn on the tape cassette motor and wait for the end of the synchronizing header from the cassette player

Entry Point: 1499h = 5273d

Input: From the cassette player

Output: None

BASIC Example:

```
CALL 5273
```

Special Comments: None

Summary

In this chapter we have examined the operation of the cassette tape interface on the Model 100. We have seen how the cassette motor is controlled through bit 3 of port E8h = 232d and how the data lines to and from the cassette are connected to the serial data lines of the 8085 CPU. We have discussed how to control the motor circuit for other purposes than simply controlling a cassette player/recorder. We have also studied the low-level ROM routines for reading from and writing to the cassette player/recorder.

BASIC Function Addresses

Address 40h = 64d

Address	Function
3407h = 13,319d	SGN
3654h = 13,908d	INT
33F2h = 13,298d	ABS
2B4Ch = 11,084d	FRE
1100h = 4,352d	INP
10C8h = 4,296d	LPOS
10CEh = 4,302d	POS
305Ah = 12,378d	SQR
313Eh = 12,606d	RND
2FCFh = 12,239d	LOG
30A4h = 12,452d	EXP
2EEFh = 12,015d	COS
2F09h = 12,041d	SIN
2F58h = 12,120d	TAN
2F71h = 12,145d	ATN
1284h = 4,740d	PEEK
1889h = 6,281d	EOF
506Dh = 20,589d	LOC
506Bh = 20,587d	LOF
3501h = 13,569d	CINT
352Ah = 13,610d	CSNG
35BAh = 13,754d	CDBL
3645h = 13,893d	FIX
2943h = 10,563d	LEN
273Ah = 10,042d	STR$
2A07h = 10,759d	VAL
294Fh = 10,575d	ASC
295Fh = 10,591d	CHR$
298Eh = 10,638d	SPACE$
29ABh = 10,667d	LEFT$
29DCh = 10,716d	RIGHT$
29E6h = 10,726d	MID$

BASIC Keywords

Address 80h = 128d

Keyword	Token Value
END	80h = 128d
FOR	81h = 129d
NEXT	82h = 130d
DATA	83h = 131d
INPUT	84h = 132d
DIM	85h = 133d
READ	86h = 134d
LET	87h = 135d
GOTO	88h = 136d
RUN	89h = 137d
IF	8Ah = 138d
RESTORE	8Bh = 139d
GOSUB	8Ch = 140d
RETURN	8Dh = 141d
REM	8Eh = 142d
STOP	8Fh = 143d
WIDTH	90h = 144d
ELSE	91h = 145d
LINE	92h = 146d
EDIT	93h = 147d
ERROR	94h = 148d
RESUME	95h = 149d
OUT	96h = 150d
ON	97h = 151d
DSKO$	98h = 152d
OPEN	99h = 153d
CLOSE	9Ah = 154d
LOAD	9Bh = 155d
MERGE	9Ch = 156d
FILES	9Dh = 157d
SAVE	9Eh = 158d
LFILES	9Fh = 159d

LPRINT	A0h = 160d
DEF	A1h = 161d
POKE	A2h = 162d
PRINT	A3h = 163d
CONT	A4h = 164d
LIST	A5h = 165d
LLIST	A6h = 166d
CLEAR	A7h = 167d
CLOAD	A8h = 168d
CSAVE	A9h = 169d
TIME$	AAh = 170d
DATE$	ABh = 171d
DAY$	ACh = 172d
COM	ADh = 173d
MDM	AEh = 174d
KEY	AFh = 175d
CLS	B0h = 176d
BEEP	B1h = 177d
SOUND	B2h = 178d
LCOPY	B3h = 179d
PSET	B4h = 180d
PRESET	B5h = 181d
MOTOR	B6h = 182d
MAX	B7h = 183d
POWER	B8h = 184d
CALL	B9h = 185d
MENU	BAh = 186d
IPL	BBh = 187d
NAME	BCh = 188d
KILL	BDh = 189d
SCREEN	BEh = 190d
NEW	BFh = 191d
TAB(	C0h = 192d
TO	C1h = 193d
USING	C2h = 194d
VARPTR	C3h = 195d
ERL	C4h = 196d
ERR	C5h = 197d
STRING$	C6h = 198d
INSTR	C7h = 199d
DSKI$	C8h = 200d
INKEY$	C9h = 201d

CSRLIN	CAh = 202d
OFF	CBh = 203d
HIMEM	CCh = 204d
THEN	CDh = 205d
NOT	CEh = 206d
STEP	CFh = 207d
+	D0h = 208d
-	D1h = 209d
*	D2h = 210d
/	D3h = 211d
^	D4h = 212d
AND	D5h = 213d
OR	D6h = 214d
XOR	D7h = 215d
EQV	D8h = 216d
IMP	D9h = 217d
MOD	DAh = 218d
\	DBh = 219d
>	DCh = 220d
=	DDh = 221d
<	DEh = 222d
SGN	DFh = 223d
INT	E0h = 224d
ABS	E1h = 225d
FRE	E2h = 226d
INP	E3h = 227d
LPOS	E4h = 228d
POS	E5h = 229d
SQR	E6h = 230d
RND	E7h = 231d
LOG	E8h = 232d
EXP	E9h = 233d
COS	EAh = 234d
SIN	EBh = 235d
TAN	ECh = 236d
ATN	EDh = 237d
PEEK	EEh = 238d
EOF	EFh = 239d
LOC	F0h = 240d
LOF	F1h = 241d
CINT	F2h = 242d
CSNG	F3h = 243d

CDBL	F4h = 244d
FIX	F5h = 245d
LEN	F6h = 246d
STR$	F7h = 247d
VAL	F8h = 248d
ASC	F9h = 249d
CHR$	FAh = 250d
SPACE$	FBh = 251d
LEFT$	FCh = 252d
RIGHT$	FDh = 253d
MID$	FEh = 254d

BASIC Command Addresses

Address 262h = 610d

Address	Command
409Fh = 16,543d	END
0726h = 1,830d	FOR
4174h = 16,756d	NEXT
099Eh = 2,462d	DATA
0CA3h = 3,235d	INPUT
478Bh = 18,315d	DIM
0CD9h = 3,289d	READ
09C3h = 2,499d	LET
0936h = 2,358d	GOTO
090Fh = 2,319d	RUN
0B1Ah = 2,842d	IF
407Fh = 16,511d	RESTORE
091Eh = 2,334d	GOSUB
0966h = 2,406d	RETURN
09A0h = 2,464d	REM
409Ah = 16,538d	STOP
1DC3h = 7,619d	WIDTH
09A0h = 2,464d	ELSE
0C45h = 3,141d	LINE
5E51h = 24,145d	EDIT
0B0Fh = 2,831d	ERROR
0AB0h = 2,736d	RESUME
110Ch = 4,364d	OUT
0A2Fh = 2,607d	ON
5071h = 20,593d	DSKO$
4CCBh = 19,659d	OPEN
4E28h = 20,008d	CLOSE

4D70h = 19,824d	LOAD
4D71h = 19,825d	MERGE
1F3Ah = 7,994d	FILES
4DCFh = 19,919d	SAVE
506Fh = 20,591d	LFILES
0B4Eh = 2,894d	LPRINT
0872h = 2,162d	DEF
128Bh = 4,747d	POKE
0B56h = 2,902d	PRINT
40DAh = 16,602d	CONT
1140h = 4,416d	LIST
113Bh = 4,411d	LLIST
40F9h = 16,633d	CLEAR
2377h = 9,079d	CLOAD
2280h = 8,832d	CSAVE
19ABh = 6,571d	TIME$
19BDh = 6,589d	DATE$
19F1h = 6,641d	DAY$
1A9Eh = 6,814d	COM
1A9Eh = 6,814d	MDM
1BB8h = 7,096d	KEY
4231h = 16,945d	CLS
4229h = 16,937d	BEEP
1DC5h = 7,621d	SOUND
1E5Eh = 7,774d	LCOPY
1C57h = 7,255d	PSET
1C66h = 7,270d	PRESET
1DECh = 7,660d	MOTOR
7F0Bh = 32,523d	MAX
1419h = 5,145d	POWER
1DFAh = 7,674d	CALL
5797h = 22,423d	MENU
1A78h = 6,776d	IPL
2037h = 8,247d	NAME
1F91h = 8,081d	KILL
1E22h = 7,714d	SCREEN
20FEh = 8,446d	NEW

Operator Priorities for Binary Operations

Address 2E2h = 738d

Priority Number	Operation
79h = 121d	+
79h = 121d	−
7Ch = 124d	*
7Ch = 124d	/
7Fh = 127d	^
50h = 80d	AND
46h = 70d	OR
3Ch = 60d	XOR
32h = 50d	EQV
28h = 40d	IMP
7Ah = 122d	MOD
7Bh = 123d	\

Some Numerical Conversion Routines

Address 2EEh = 750d

Address	Operation
35BAh = 13,754d	CDBL (Convert to Double Precision)
0000h = 0d	none
3501h = 13,569d	CINT (Convert to Integer)
35D9h = 13,785d	check for integer type
352Ah = 13,610d	CSNG (Convert to Single Precision)

Binary Operations for Double Precision

Address 2F8h = 760d

Address	Operation
2B78h = 11,128d	+
2B69h = 11,113d	−
2CFFh = 11,518d	*
2DC7h = 11,719d	/
3D8Eh = 15,758d	^
34FAh = 13,562d	comparisons

Binary Operations for Single Precision

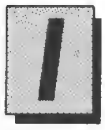
Address 304h = 772d

Address	Operation
37F4h = 14,324d	+
37FDh = 14,333d	−
3803h = 14,339d	*
380Eh = 14,350d	/
3D7Fh = 15,743d	^
3498h = 13,464d	comparisons

Binary Operations for Integers

Address 310h = 784d

Address	Operation
3704h = 14,084d	+
36F8h = 14,072d	−
3725h = 14,117d	*
0F0Dh = 3,853d	/
3DF7h = 15,863d	^
34C2h = 13,506d	comparisons



Error Codes

Address 31Ch = 796d

Symbol	Code	Explanation
NF	1	NEXT WITHOUT FOR
SN	2	SYNTAX ERROR
RG	3	RETURN WITHOUT GOSUB
OD	4	OUT OF DATA
FC	5	ILLEGAL FUNCTION CALL
OV	6	OVERFLOW
OM	7	OUT OF MEMORY
UL	8	UNDEFINED LINE
BS	9	BAD SUBSCRIPT
DD	10	DOUBLE DIMENSIONED ARRAY
/0	11	DIVISION BY ZERO
ID	12	ILLEGAL DIRECT
TM	13	TYPE MISMATCH
OS	14	OUT OF STRING SPACE
LS	15	STRING TOO LONG
ST	16	STRING FORMULA TOO COMPLEX
CN	17	CAN'T CONTINUE
IO	18	ERROR
NR	19	NO RESUME
RW	20	RESUME WITHOUT ERROR
UE	21	UNDEFINED ERROR
MO	22	MISSING OPERAND
IE	50	UNDEFINED ERROR
BN	51	BAD FILE NUMBER
FF	52	FILE NOT FOUND
AO	53	ALREADY OPEN
EF	54	INPUT PAST END OF FILE
NM	55	BAD FILE NAME
DS	56	DIRECT STATEMENT IN FILE
FL	57	UNDEFINED ERROR
CF	58	FILE NOT OPEN

BASIC Error Routines

Address	Code	Explanation
446h = 1,094d	02h = 2d	SYNTAX ERROR
449h = 1,097d	0Bh = 11d	DIVISION BY 0
44Ch = 1,100d	01h = 1d	NEXT WITHOUT FOR
44Fh = 1,103d	0Ah = 10d	DOUBLE DIM ARRAY
452h = 1,106d	14h = 20d	RESUME WITHOUT ERROR
455h = 1,109d	06h = 6d	OVERFLOW
458h = 1,112d	16h = 22d	MISSING OPERAND
45Bh = 1,115d	0Dh = 13d	TYPE MISMATCH
504Eh = 20,558d	37h = 55d	BAD FILE NAME
5051h = 20,561d	35h = 53d	ALREADY OPEN
5054h = 20,564d	38h = 56d	DIRECT STATEMENT IN FILE
5057h = 20,567d	34h = 52d	FILE NOT FOUND
505Ah = 20,570d	3Ah = 58d	FILE NOT OPEN
505Dh = 20,573d	33h = 51d	BAD FILE NUMBER
5060h = 20,576d	32h = 50d	UNDEFINED ERROR
5063h = 20,579d	36h = 54d	INPUT PAST END OF FILE
5066h = 20,582d	39h = 57d	UNDEFINED ERROR

Control Characters for the Model 100

Address 438Ah = 17,290d

ASCII Code	Address of Routine	Function
07h = 7d	7662h = 30,306d	BELL
08h = 8d	4461h = 17,505d	BACKSPACE
09h = 9d	4480h = 17,536d	TAB
0Ah = 10d	4494h = 17,556d	LF
0Bh = 11d	44A8h = 17,576d	HOME
0Ch = 12d	4548h = 17,736d	FF
0Dh = 13d	44AAh = 17,578d	CR
1Bh = 27d	43B2h = 17,330d	ESC

Routines for Escape Sequences

Address 43B8h = 17,336d

ASCII code	Character after ESC Symbol	Address of Routine	Function
6Ah = 106d	j	4548h = 17,736d	ERASE SCREEN
45h = 69d	E	4548h = 17,736d	ERASE SCREEN
4Bh = 75d	K	4537h = 17,719d	ERASE TO END OF LINE
4Ah = 74d	J	454Eh = 17,742d	CLEAR TO END OF SCREEN
6Ch = 108d	l	4535h = 17,717d	ERASE LINE
4Ch = 76d	L	44EAh = 17,642d	INSERT BLANK LINE
4Dh = 77d	M	44C4h = 17,604d	DELETE LINE
59h = 89d	Y	43AFh = 17,327d	DIRECT CURSOR ADDRESSING
41h = 65d	A	4469h = 17,513d	CURSOR UP
42h = 66d	B	446Eh = 17,518d	CURSOR DOWN
43h = 67d	C	4453h = 17,491d	CURSOR RIGHT
44h = 68d	D	445Ch = 17,500d	CURSOR LEFT
48h = 72d	H	44A8h = 17,576d	HOME
70h = 112d	p	4431h = 17,457d	SET REVERSE CHAR
71h = 113d	q	4432h = 17,458d	TURN OFF REVERSE CHAR

50h = 80d	P	44AFh = 17,583d	TURN ON CURSOR
51h = 81d	Q	44BAh = 17,594d	TURN OFF CURSOR
54h = 84d	T	4439h = 17,465d	SET SYSTEM LINE
55h = 85d	U	4437h = 17,463d	RESET SYSTEM LINE
56h = 86d	V	443Fh = 17,471d	LOCK DISPLAY
57h = 87d	W	4440h = 17,472d	UNLOCK DISPLAY
58h = 88d	X	444Ah = 17,482d	

Special Screen Routines for the Model 100

Address	Entry Condition	Function
20h = 32d	A has ASCII code	Print a character
4222h = 16,930d	None	Print CR/LF
4229h = 16,937d	None	Beep
422Dh = 16,941d	None	Home cursor
4231h = 16,945d	None	Clear the screen
4235h = 16,949d	None	Lock system line
423Ah = 16,954d	None	Unlock system line
423Fh = 16,959d	None	Disable scrolling
4244h = 16,964d	None	Enable scrolling
4249h = 16,969d	None	Turn on cursor
424Eh = 16,974d	None	Turn off cursor
4253h = 16,979d	None	Delete line at cursor
4258h = 16,984d	None	Insert blank line
425Dh = 16,989d	None	Erase to end of line
4262h = 16,994d	None	Send ESC X
4269h = 17,001d	None	Set reverse character
426Eh = 17,006d	None	Turn off reverse char
4270h = 17,008d	A has ESC code	Send escape sequence
4277h = 17,015d	None	Send cursor to lower left corner of screen
427Ch = 17,020d	H = column (1-40) L = row (1-8)	Set cursor position
428Ah = 17,034d	None	Erase label line
42A5h = 17,061d	HL = address of function table	Set and display function table

LCD Data for Character Positions

Address 7551h = 30,033d

Column	Send to ports B9h = 185d and BAh = 186d	Send to port FEh = 254d
UPPER HALF OF DISPLAY:		
1	0001h	00h = 0d
2	0001h	06h = 6d
3	0001h	0Ch = 12d
4	0001h	12h = 18d
5	0001h	18h = 24d
6	0001h	1Eh = 30d
7	0001h	24h = 36d
8	0001h	2Ah = 42d
9	0001h	30h = 48d
10	0002h	04h = 4d
11	0002h	0Ah = 10d
12	0002h	10h = 16d
13	0002h	16h = 22d
14	0002h	1Ch = 28d
15	0002h	22h = 34d
16	0002h	28h = 40d
17	0002h	2Eh = 46d
18	0004h	02h = 2d
19	0004h	08h = 8d
20	0004h	0Eh = 14d
21	0004h	14h = 20d
22	0004h	1Ah = 26d
23	0004h	20h = 32d
24	0004h	26h = 38d

25	0004h	2Ch = 44d
26	0008h	00h = 0d
27	0008h	06h = 6d
28	0008h	0Ch = 12d
29	0008h	12h = 18d
30	0008h	18h = 24d
31	0008h	1Eh = 30d
32	0008h	24h = 36d
33	0008h	2Ah = 42d
34	0008h	30h = 48d
35	0010h	04h = 4d
36	0010h	0Ah = 10d
37	0010h	10h = 16d
38	0010h	16h = 22d
39	0010h	1Ch = 28d
40	0010h	22h = 34d

LOWER HALF OF DISPLAY:

1	0020h	00h = 0d
2	0020h	06h = 6d
3	0020h	0Ch = 12d
4	0020h	12h = 18d
5	0020h	18h = 24d
6	0020h	1Eh = 30d
7	0020h	24h = 36d
8	0020h	2Ah = 42d
9	0020h	30h = 48d
10	0040h	04h = 4d
11	0040h	0Ah = 10d
12	0040h	10h = 16d
13	0040h	16h = 22d
14	0040h	1Ch = 28d
15	0040h	22h = 34d
16	0040h	28h = 40d
17	0040h	2Eh = 46d
18	0080h	02h = 2d
19	0080h	08h = 8d
20	0080h	0Eh = 14d
21	0080h	14h = 20d
22	0080h	1Ah = 26d
23	0080h	20h = 32d
24	0080h	26h = 38d

25	0080h	2Ch = 44d
26	0100h	00h = 0d
27	0100h	06h = 6d
28	0100h	0Ch = 12d
29	0100h	12h = 18d
30	0100h	18h = 24d
31	0100h	1Eh = 30d
32	0100h	24h = 36d
33	0100h	2Ah = 42d
34	0100h	30h = 48d
35	0200h	04h = 4d
36	0200h	0Ah = 10d
37	0200h	10h = 16d
38	0200h	16h = 22d
39	0200h	1Ch = 28d
40	0200h	22h = 34d
FINISHING PATTERN:		
	03FFh	01h = 1d

ASCII Tables for Regular Keys

Lowercase 7BF1h = 31,729d	Uppercase 7C1Dh = 31,773d
7Ah = 122d z	5Ah = 90d (SHIFT) Z
78h = 120d x	58h = 88d (SHIFT) X
63h = 99d c	43h = 67d (SHIFT) C
76h = 118d v	56h = 86d (SHIFT) V
62h = 98d b	42h = 66d (SHIFT) B
6Eh = 110d n	4Eh = 78d (SHIFT) N
6Dh = 109d m	4Dh = 77d (SHIFT) M
6Ch = 108d l	4Ch = 76d (SHIFT) L
61h = 97d a	41h = 65d (SHIFT) A
73h = 115d s	53h = 83d (SHIFT) S
64h = 100d d	44h = 68d (SHIFT) D
66h = 102d f	46h = 70d (SHIFT) F
67h = 103d g	47h = 71d (SHIFT) G
68h = 104d h	48h = 72d (SHIFT) H
6Ah = 106d j	4Ah = 74d (SHIFT) J
6Bh = 107d k	4Bh = 75d (SHIFT) K
71h = 113d q	51h = 81d (SHIFT) Q
77h = 119d w	57h = 87d (SHIFT) W
65h = 101d e	45h = 69d (SHIFT) E
72h = 114d r	52h = 82d (SHIFT) R
74h = 116d t	54h = 84d (SHIFT) T
79h = 121d y	59h = 89d (SHIFT) Y
75h = 117d u	55h = 85d (SHIFT) U
69h = 105d i	49h = 73d (SHIFT) I
6Fh = 111d o	4Fh = 79d (SHIFT) O
70h = 112d p	50h = 80d (SHIFT) P
5Bh = 91d [	5Dh = 93d (SHIFT)]
3Bh = 59d ;	3Ah = 58d (SHIFT) :
27h = 39d '	22h = 34d (SHIFT) "

2Ch = 44d ,
2Eh = 46d .
2Fh = 47d /
31h = 49d 1
32h = 50d 2
33h = 51d 3
34h = 52d 4
35h = 53d 5
36h = 54d 6
37h = 55d 7
38h = 56d 8
39h = 57d 9
30h = 48d 0
2Dh = 45d −
3Dh = 61d =

3Ch = 60d [SHIFT] <
3Eh = 62d [SHIFT] >
3Fh = 63d [SHIFT] ?
21h = 33d [SHIFT] !
40h = 64d [SHIFT] @
23h = 35d [SHIFT] #
24h = 36d [SHIFT] $
25h = 37d [SHIFT] %
5Eh = 94d [SHIFT] ^
26h = 38d [SHIFT] &
2Ah = 42d [SHIFT] *
28h = 40d [SHIFT] (
29h = 41d [SHIFT])
5Fh = 95d [SHIFT] _
2Bh = 43d [SHIFT] +

Unshifted GRPH
7C49h = 31,817d

00h = 0d [GRPH] z
83h = 131d [GRPH] x
84h = 132d [GRPH] c
00h = 0d [GRPH] v
95h = 149d [GRPH] b
96h = 150d [GRPH] n
81h = 129d [GRPH] m
9Ah = 154d [GRPH] l
85h = 133d [GRPH] a
8Bh = 139d [GRPH] s
00h = 0d [GRPH] d
82h = 130d [GRPH] f
00h = 0d [GRPH] g
86h = 134d [GRPH] h
00h = 0d [GRPH] j
9Bh = 155d [GRPH] k
93h = 147d [GRPH] q
94h = 148d [GRPH] w
8Fh = 143d [GRPH] e
89h = 137d [GRPH] r
87h = 135d [GRPH] t
90h = 144d [GRPH] y
91h = 145d [GRPH] u

SHIFT GRPH
7C75h = 31,861d

E0h = 224d [SHIFT] [GRPH] Z
EFh = 239d [SHIFT] [GRPH] X
FFh = 255d [SHIFT] [GRPH] C
00h = 0d [SHIFT] [GRPH] V
00h = 0d [SHIFT] [GRPH] B
00h = 0d [SHIFT] [GRPH] N
F6h = 246d [SHIFT] [GRPH] M
F9h = 249d [SHIFT] [GRPH] L
EBh = 235d [SHIFT] [GRPH] A
ECh = 236d [SHIFT] [GRPH] S
EDh = 237d [SHIFT] [GRPH] D
EEh = 238d [SHIFT] [GRPH] F
FDh = 253d [SHIFT] [GRPH] G
FBh = 251d [SHIFT] [GRPH] H
F4h = 244d [SHIFT] [GRPH] J
FAh = 250d [SHIFT] [GRPH] K
E7h = 231d [SHIFT] [GRPH] Q
E8h = 232d [SHIFT] [GRPH] W
E9h = 233d [SHIFT] [GRPH] E
EAh = 234d [SHIFT] [GRPH] R
FCh = 252d [SHIFT] [GRPH] T
FEh = 254d [SHIFT] [GRPH] Y
F0h = 240d [SHIFT] [GRPH] U

8Eh = 142d (GRPH) i
98h = 152d (GRPH) o
80h = 128d (GRPH) p
60h = 96d (GRPH) [
92h = 146d (GRPH) ;
8Ch = 140d (GRPH) '
99h = 153d (GRPH) ,
97h = 151d (GRPH) .
8Ah = 138d (GRPH) /
88h = 136d (GRPH) 1
9Ch = 156d (GRPH) 2
9Dh = 157d (GRPH) 3
9Eh = 158d (GRPH) 4
9Fh = 159d (GRPH) 5
B4h = 180d (GRPH) 6
B0h = 176d (GRPH) 7
A3h = 163d (GRPH) 8
7Bh = 123d (GRPH) 9
7Dh = 125d (GRPH) 0
5Ch = 92d (GRPH) −
8Dh = 141d (GRPH) =

Unshifted CODE
7CA1h = 31,905d

CEh = 206d (CODE) z
A1h = 161d (CODE) x
A2h = 162d (CODE) c
BDh = 189d (CODE) v
00h = 0d (CODE) b
CDh = 205d (CODE) n
00h = 0d (CODE) m
CAh = 202d (CODE) l
B6h = 182d (CODE) a
A9h = 169d (CODE) s
BBh = 187d (CODE) d
00h = 0d (CODE) f
00h = 0d (CODE) g
00h = 0d (CODE) h
CBh = 203d (CODE) j
C9h = 201d (CODE) k
C8h = 200d (CODE) q

F3h = 243d (SHIFT) (GRPH) I
F2h = 242d (SHIFT) (GRPH) O
F1h = 241d (SHIFT) (GRPH) P
7Eh = 126d (SHIFT) (GRPH)]
F5h = 245d (SHIFT) (GRPH) :
00h = 0d (SHIFT) (GRPH) "
F8h = 248d (SHIFT) (GRPH) <
F7h = 247d (SHIFT) (GRPH) >
00h = 0d (SHIFT) (GRPH) ?
E1h = 225d (SHIFT) (GRPH) !
E2h = 226d (SHIFT) (GRPH) @
E3h = 227d (SHIFT) (GRPH) #
E4h = 228d (SHIFT) (GRPH) $
E5h = 229d (SHIFT) (GRPH) %
E6h = 230d (SHIFT) (GRPH) ^
00h = 0d (SHIFT) (GRPH) &
00h = 0d (SHIFT) (GRPH) *
00h = 0d (SHIFT) (GRPH) (
00h = 0d (SHIFT) (GRPH))
7Ch = 124d (SHIFT) (GRPH) _
00h = 0d (SHIFT) (GRPH) +

SHIFT CODE
7CCDh = 31,949d

00h = 0d (SHIFT) (CODE) Z
DFh = 223d (SHIFT) (CODE) X
ABh = 171d (SHIFT) (CODE) C
DEh = 222d (SHIFT) (CODE) V
00h = 0d (SHIFT) (CODE) B
00h = 0d (SHIFT) (CODE) N
A5h = 165d (SHIFT) (CODE) M
DAh = 218d (SHIFT) (CODE) L
B1h = 177d (SHIFT) (CODE) A
B9h = 185d (SHIFT) (CODE) S
D7h = 215d (SHIFT) (CODE) D
BFh = 191d (SHIFT) (CODE) F
00h = 0d (SHIFT) (CODE) G
00h = 0d (SHIFT) (CODE) H
DBh = 219d (SHIFT) (CODE) J
D9h = 217d (SHIFT) (CODE) K
D8h = 216d (SHIFT) (CODE) Q

00h = 0d (CODE) w
C6h = 198d (CODE) e
00h = 0d (CODE) r
00h = 0d (CODE) t
CCh = 204d (CODE) y
B8h = 184d (CODE) u
C7h = 199d (CODE) i
B7h = 183d (CODE) o
ACh = 172d (CODE) p
B5h = 181d (CODE) [
ADh = 173d (CODE) ;
A0h = 160d (CODE) '
BCh = 188d (CODE) ,
CFh = 207d (CODE) .
AEh = 174d (CODE) /
C0h = 192d (CODE) 1
00h = 0d (CODE) 2
C1h = 193d (CODE) 3
00h = 0d (CODE) 4
00h = 0d (CODE) 5
00h = 0d (CODE) 6
C4h = 196d (CODE) 7
C2h = 194d (CODE) 8
C3h = 195d (CODE) 9
AFh = 175d (CODE) 0
C5h = 197d (CODE) −
BEh = 190d (CODE) =

00h = 0d (SHIFT) (CODE) W
D6h = 214d (SHIFT) (CODE) E
AAh = 170d (SHIFT) (CODE) R
BAh = 186d (SHIFT) (CODE) T
DCh = 220d (SHIFT) (CODE) Y
B3h = 179d (SHIFT) (CODE) U
D5h = 213d (SHIFT) (CODE) I
B2h = 178d (SHIFT) (CODE) O
00h = 0d (SHIFT) (CODE) P
00h = 0d (SHIFT) (CODE)]
00h = 0d (SHIFT) (CODE) :
A4h = 164d (SHIFT) (CODE) "
DDh = 221d (SHIFT) (CODE) <
00h = 0d (SHIFT) (CODE) >
00h = 0d (SHIFT) (CODE) ?
D0h = 208d (SHIFT) (CODE) !
00h = 0d (SHIFT) (CODE) @
D1h = 209d (SHIFT) (CODE) #
00h = 0d (SHIFT) (CODE) $
00h = 0d (SHIFT) (CODE) %
00h = 0d (SHIFT) (CODE) ^
D4h = 212d (SHIFT) (CODE) &
D2h = 210d (SHIFT) (CODE) *
D3h = 211d (SHIFT) (CODE) (
A6h = 166d (SHIFT) (CODE))
A7h = 167d (SHIFT) (CODE) _
A8h = 168d (SHIFT) (CODE) +

ASCII Table for NUM Key

Memory Address	Regular Key	NUM Pad Key	
7CF9h = 31,993d	6Dh = 109d	30h = 48d	0
7CFBH = 31,995d	6Ah = 106d	31h = 49d	1
7CFDh = 31,997d	6Bh = 107d	32h = 50d	2
7CFFh = 31,999d	6Ch = 108d	33h = 51d	3
7DD1h = 32,001d	75h = 117d	34h = 52d	4
7DO3h = 32,003d	69h = 105d	35h = 53d	5
7DO5h = 32,005d	6Fh = 111d	36h = 54d	6
	Byte	Byte	

ASCII Tables for Special Keys

7DO7h = 32,007d

01h = 1d	←
06h = 6d	→
14h = 20d	↑
02h = 2d	↓
20h = 32d	SPACE
7Fh = 127d	DEL
09h = 9d	TAB
1Bh = 27d	ESC
8Bh = 139d	PASTE
88h = 136d	LABEL
8Ah = 138d	PRINT
0Dh = 13d	ENTER
80h = 128d	F1
81h = 129d	F2
82h = 130d	F3
83h = 131d	F4
84h = 132d	F5
85h = 133d	F6
86h = 134d	F7
87h = 135d	F8

7D1Bh = 32,027d

1Dh = 29d	SHIFT ←
1Ch = 28d	SHIFT →
1Eh = 30d	SHIFT ↑
1Fh = 31d	SHIFT ↓
20h = 32d	SHIFT SPACE
08h = 8d	SHIFT BKSP
09h = 9d	SHIFT TAB
1Bh = 27d	SHIFT ESC
8Bh = 139d	SHIFT PASTE
88h = 136d	SHIFT LABEL
89h = 137d	SHIFT PRINT
0Dh = 13d	SHIFT ENTER
80h = 128d	SHIFT F1
81h = 129d	SHIFT F2
82h = 130d	SHIFT F3
83h = 131d	SHIFT F4
84h = 132d	SHIFT F5
85h = 133d	SHIFT F6
86h = 134d	SHIFT F7
87h = 135d	SHIFT F8

Index

6402 UART, 20, 34
8085 CPU, 23, 26, 28, 41
8085 microprocessor, 22
80C85 CPU, 19
8155 PIO, 19, 20, 22, 31, 35, 178, 196

ADDRSS, 4, 38, 40, 72, 77-78
ALE, 26
ASCII code, 103
ASCII file(s), 72, 74, 79, 210
Accumulator, 23, 67
Address, 7, 16, 24
  bus, 24
  finder, 60, 62
  lines, 26
  selection, 27
  tables, 51
Addressing space(s), 27, 28
Area filling, 100
Arithmetic logic unit, 22, 23
Assembly language, 7, 8
Assembly-language mnemonics, 22
Automatic power shutoff, 115
Automatic power-off, 145-46

BASIC, 1, 2, 17, 38, 52, 70, 72, 74
  commands, 51
  interpreter, 2, 40, 51, 57
  interrupt, 153
  keywords, 51
  program files, 74
BEEP, 76, 196, 198, 199
BREAK, 191
BRKCHK, 166-67
Background task, 50, 106, 111, 115, 121, 142-47, 153, 156-65
Bank, 88, 90
Bank selector, 89-90
Bar code reader, 20, 49
Bar code reader interface, 4, 41
Binary operations, 52, 69
Block transfer, 137
Boxes, 91
Box-fill, 101, 102
Buses, 19
Bus interfacing, 22, 24
Bus system, 24
Byte plotting, 112-13

CALL, 17
CASIN, 215
CLK, 26
CLSCOM, 180
CMOS, 22, 31
COM ON, 174
COM STOP, 174
CONN, 183
CPU, 20, 22, 24
CPU instruction(s), 8, 14
CPU registers, 22
CSOUT, 210
CTOFF, 207
CTON, 212, 216, 206
CTS, 174
Cassette recorder, 33, 208
Chip
  enable, 27
  select, 27
Circular buffer, 186-89
Clock, 22, 88
  command, 121
  speed, 22
Cold start, 80
Control, 24, 31, 33
  bus, 28
  characters, 107
  line, 30
Cursor, 40, 71, 78
  addressing, 107
  blink, 91, 113-14
  control, 76

DATAW, 208-9, 211
DATE$, 128
DAY$, 131, 136
DEFDBL, 62
DEFINT, 62
DEFSNG, 62
DI instruction, 49
DIAL, 181
DISC, 183
DSR, 174
DTR, 174
Data, 24
  bus, 24
  lines, 26
  type, 45
Decoding, 28-30
Dialing, 180, 181, 184
Disassembler, 7, 8
Disk drive/video interface, 20
Double precision, 67
  format, 52
  numbers, 63, 64

Editing, 78
Entry point(s), 15, 16, 38, 41
Error, 54
  codes, 54
  designators, 52
  entry points, 54
  routine, 54

FOR statement, 58
File(s), 2
  directory, 2, 72-74
  saving and loading, 2
  system, 2
  type, 72, 74

H, 14
Hook(s), 47, 104
  table, 104
Horizontal LCD drivers, 85-88

IN, 29
INITIO, 80, 81
INLIN, 55
INP, 53
INPUT, 88
INT, 51
INTA, 26
INTR, 26
INZCOM, 177-80
IO/M, 26
Instruction
  register, 24
  decoding, 22, 24
Interrupt(s), 22, 26, 32, 139
  7.5, 96
  OFF, 138
  ON, 138
  STOP, 139
  control, 22
  counter, 142
  routines, 41
  status byte, 139

KEY OFF, 153-56
KEY ON, 153-56
KEY STOP, 153-56
KEYBOARD MATRIX, 152
KEYX, 167-68
KYREAD, 165-66
Keyboard, 4, 22, 31, 35, 52, 55, 57, 88, 148-68
  input, 153
  matrix, 149
  scanning, management, 157

LCD RAM, 108-10
LCD, 15, 16, 22, 25, 31, 72, 153
  screen, 52
LDA, 29

LET, 58-69
  command, 58
LHLD, 29
LINE, 101
Liquid crystal display, 15, 34, 82-114

μPD 1990, 20
μPD 1990 board, 19
MAKTXT, 79
MDM OFF, 174
MDM ON, 174
MDM STOP, 174
MENU, 38, 40, 71-77, 79
MODEM, 25
MOV, 29
MUSIC, 201
Memory, 26, 28
  power switch, 25
  space, 1
Microprocessor chip, 24
Modem, 3, 4, 174, 181
Motor control, 203, 205

ON COM, 142
ON COM GOSUB, 174
ON KEY, 142
ON KEY GOSUB, 153
ON MDM, 142
ON MDM GOSUB, 174
ON TIME$, 115, 139, 140
ON TIME$...GOSUB, 136, 137
ON TIME$ Interrupt, 146
ON...INTERRUPT/GOSUB, 154
OUT, 15, 29, 53
Ok, 54
Output, 16, 31
  ports, 15

PC, 23
PEEK, 5, 51
PIO, 31
PIO bus, 22
PLOT, 95-99, 101
PRESET, 88, 93-94, 95
PSET, 88, 93, 95
Parallel, 22, 170
  I/O, 31
  transfer, 116, 127
Plane of vibration, 83
Points, 91
Pointer(s), 2, 23
Port A, 31
Port B, 31
Port C, 31, 33
Port decoder, 31
Printer, 19, 22, 31
  interface, 4, 35
Priority, 66
Program counter, 23
Protection code, 72

RAM RST, 26
RCVX, 189, 190
RD, 26
REMOTE, 206-7
RESET, 26, 41
RIM, 22, 24, 204
ROM, 1, 16, 20, 26, 38, 41, 52
  file(s), 2, 72, 74
RS-232, 25, 169
RS-232C, 171, 174, 181
  communications, 69, 70
  serial port, 3, 4
RST, 41
RST 0, 41, 42
RST 1, 42
RST 2, 43, 69
RST 3, 43, 44
RST 4, 44, 103
RST 5, 45
RST 5.5, 24, 49
RST 6, 46
RST 6.5, 24, 50
RST 7, 47, 104
RST 7.5, 24, 50
RV232C, 190-91
Read
  cassette data, 214
  LCD bytes, 98
  time and date, 125-27
Real time clock, 25, 31, 33, 34, 41, 57, 115-47

S0, 26
S1, 26
SCHDL, 38
SCHEDL, 4, 40, 77-78
SD232C, 191, 192
SENDCQ, 194
SENDCS, 195
SHLD, 29
SID, 204, 214
SIM, 22, 24, 144-45, 204, 209
SNDCOM, 191-93
SOD, 204, 209
SOUND, 196, 201
SOUND OFF, 214
STA, 29
SYNCR, 216
SYNCW, 211, 212
Serial, 170-74
  interrupt service routine, 186
  output data (SOD), 24
  communications, 41, 50, 71, 169
  control, 22, 24
  transfer, 116, 127
  transmission, 193
Set date, 135
Set time, 133
Set the clock, 121
Shifting operations, 23
Sign, 46, 64
Sound, 196-207
  delay, 200
Special comments, 17
Status, 26, 31, 33
Stopwatch program, 144
String, 63, 64
Strobe(s), 33, 34, 36, 127-28
Synchronizing header, 211, 212, 216

TC55188F-25 chips, 27
TELCOM, 3, 4, 38, 40, 69, 70, 72, 74
TERM, 70, 71
TEXT, 3, 38, 40, 72, 74, 78
TIME$, 122, 128, 132, 133
TIME$ OFF, 136, 138, 140
TIME$ ON, 136, 138, 140
TIME$ STOP, 136, 138, 140
TRAP, 24, 49
Tape cassette interface, 4, 20, 203-17
Timer, 22, 31, 32, 33
Timing, 26
  pulse, 118
  signal, 24
Timing and control, 22-24
Token(s), 51
Tokenize, 56
Trigger interrupt, 141

UART(s), 172, 174-176, 185, 196
UNPLOT, 95-99, 101
Updating the year, 147
Uploading, 17
User-defined functions, 2

VARPTR, 61
VB, 25
VDD, 25
VEE, 25
Variable(s), 2, 62
  name, 61
  type, 61-64
Vertical LCD drivers, 86-87

WR, 26
Write cassette data, 209
Write LCD bytes, 99

Y0, 26, 29, 30
Y1, 30
Y2, 30
Y3, 31
Y7, 29, 30

(0452)

Other PLUME/WAITE books on the TRS-80® Model 100:

☐ **Introducing the TRS-80® Model 100, by Diane Burns and S. Venit.** This book, intended for newcomers to the Model 100, offers simple step-by-step explanations of how to set up your Model 100 and how to use its built-in programs: TEXT, ADDRSS, SCHEDL, TELCOM, and BASIC. Specific instructions are given for connecting the Model 100 to the cassette recorder, other computers, the telephone lines, the optional disk drive/video interface, and the optional bar code reader. (255740—$15.95)

☐ **Mastering BASIC on the TRS-80® Model 100, by Bernd Enders.** An exceptionally easy-to-follow introduction to the built-in programming language on the Model 100. Also serves as a comprehensive reference guide for the advanced user. Covers all Model 100 BASIC features including graphics, sound, and file-handling. With this book and the Model 100 you can learn BASIC anywhere! (255759—$19.95)

☐ **Games and Utilities for the TRS-80® Model 100, by Ron Karr, Steven Olsen, and Robert Lafore.** A collection of powerful programs to enhance your Model 100. Enjoy fast-paced, exciting card games, arcade games, music, art, and learning games. Help yourself to practical utilities that let you count words in a text file, turn your Model 100 into a scientific calculator, show file sizes, and generally increase your Model 100's usefulness, and your own grasp of programming. (XXXXXX — $XX.XX)

☐ **Practical Finance on the TRS-80® Model 100, by S. Venit and Diane Burns.** The perfect book for anyone using the Model 100 in business: investors, real estate brokers, managers. Contains short but powerful programs to perform production planning, and access financial and other information from CompuServe® and the Dow Jones News/Retrieval® service. (255767—$15.95)

All prices higher in Canada.

To order, use the convenient coupon on the next page.

The Waite Group

COMPUTER • Z5578 • $19.95
CANADA • $24.95

PLUME
WAITE

HIDDEN POWERS OF THE TRS-80® MODEL 100

Discover how the TRS-80 Model 100 really works, and how to access its hidden features. This book explores the powerful "secret" ROM routines of the Model 100. It reveals the internal bit-level operations of the liquid crystal display, the process of sound generation, the workings of the keyboard, serial port, and other peripherals. With this book you will be able to access all of the machine's features from BASIC or assembly language.

You will learn how to scroll lines of the keyboard, print in reverse video, dial the phone directly from BASIC, and much more. Written especially for programmers who want to squeeze the last drop of power from their Model 100 software, this superior guide contains a cornucopia of fast, useful routines that will upgrade the performance of your programs, as well as your understanding of the exciting new TRS-80 Model 100.

0 25578 51857 01695
ISBN 0-452-25578-3

The Waite Group is a Sausalito, California , based producer of high-quality books on personal computing. Acknowledged as a leader in the industry, the Waite Group has written and produced over thirty titles, including such best sellers as Assembly Language Primer for the IBM PC & XT, Graphics Primer for the IBM PC, CP/M Primer, and Soul of CP/M. Internationally known and award winning, Waite Group books are distributed worldwide, and have been repackaged with the products of such major companies as Epson, Wang, Xerox, Tandy Radio-Shack, NCR and Exxon. Mr. Waite, President of the Waite Group, has been involved in the computer industry since 1976 when he bought his first Apple I computer from Steven Jobs.